URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS/LE	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

11. TITLE (Include Security Classification)
BUILDING REUSEABLE SOFTWARE COMPONENTS FOR AUTOMATED RETRIEVAL

12. PERSONAL AUTHOR(S)
Sealander, Jennie Marie

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 09/90 TO: 09/92	September 1992	100

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Reusable Software Components, Automated Retrieval, CAPS

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
The Computer Aided Prototyping System (CAPS) is designed to rapidly build prototypes of real-time sytems throug he automated retrieval of reusable software components. A critical element in achieving this goal is a mechanism fo utomated retrieval of reusable software components from a software base. There were two major objectives of thi hesis: (1) to select and prepare software components for inclusion in the CAPS software base; (2) to design and in lement a translation tool which takes an Ada specification as input and generates the prototype system description anguage (PSDL) interface required for storage and retrieval in the CAPS software base -this is necessary since for component to be usable in the CAPS software base, it must be specified in PSDL. We described the abstraction and mplementation of the selected components, introduced the translator, and demonstrated the behaviors of the transl or via examples.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Yuh-jeng Lee	22b. TELEPHONE (Include Area Code): (408) 646-2361; 22c. OFFICE SYMBOL: CS/LE

D FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited

BUILDING REUSABLE SOFTWARE COMPONENTS FOR AUTOMATED RETRIEVAL

by
Jennie Marie Sealander
Lieutenant, United States Naval Reserve
B.A., Goucher College, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

Dr. Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

The Computer Aided Prototyping System (CAPS) is designed to rapidly build prototypes of real-time systems. A critical element in achieving this goal is a mechanism for automated retrieval of reusable software components from a software base. There were two major objectives of this thesis: (1) to select and prepare software components for inclusion in the CAPS software base; (2) to design and implement a translation tool which takes an Ada specification as input and generates the prototype system description language (PSDL) interface required for storage and retrieval in the CAPS software base - this is necessary since for a component to be usable in the CAPS software base, it must be specified in PSDL. We described the abstraction and implementation of the selected components, introduced the translator, and demonstrated the behaviors of the translator via examples.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. RAPID PROTOTYPING

Developing software systems which are efficient, reliable, maintainable, and understandable is a difficult task, especially for reliable real-time systems consisting of million lines of code. The development of software tools and methods has emerged in an attempt to manage the complexity. One method designed to aid development of reliable large sytems is that of rapid prototyping. A prototype is an executable model of an intended system whose purpose is to help the system designer and customer to evaluate and validate the feasibility of the proposed system. Rapid prototyping is the process of rapidly building a prototype of a system. This allows the user to provide feedback to the designer in the development phase, reducing wasted effort in building a system which does not meet the customers needs. Thus the developement of the system follows an iterative process. The designer constructs a prototype based on the requirements, examines the execution of the prototype together with the customer, then adjusts the requirements based on feedback from the customer. The prototype is modified accordingly until both the customer and the designer agree on the requirements [10].

The key to rapid prototyping is computer-aided tools and reusable components. An efficient way to rapidly build a prototype is to construct the system out of existing software. For this to happen, rapid prototyping tools should include a library of high quality reusable components in conjunction with an automated retrieval system. A specification language is needed to specify the requirements of a system and to locate components which meet the specifications. To achieve automated retrieval, components in the library must be stored with an interface written in the specification language identifying their functionality.

The typical software life cycle for such a sytem is illustrated in Figure 1.1. The designer takes a set of requirements provided by the customer and generates specifications for the system. The prototype is realized by replacing as much of the specification as possible with reusable components. The remaining code is written manually. The goal is to

build the prototype from as many existing compoments as possible, minimizing the amount of code which needs to be written manually.

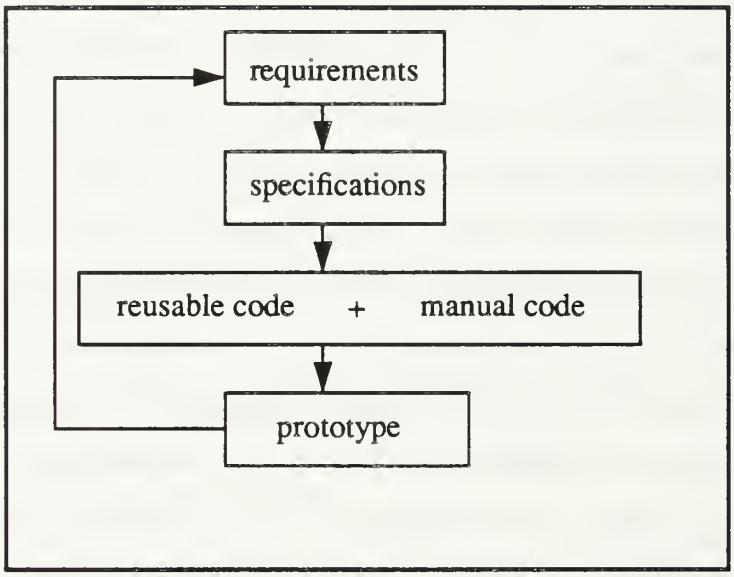

Figure 1.1: Software Life Cycle for a Prototype

B. CAPS OVERVIEW

The Computer Aided Prototyping System (CAPS) is an integrated set of computer-aided software tools being developed at the Naval Postgraduate School. The system is designed to rapidly prototype hard real-time systems [9]. The main subsystems of CAPS are illustrated in Figure 1.2. The software tools communicate by means of a specification language, the prototype system description language (PSDL). The specification language allows the designer to formally translate the customer's requirements into a high level description of the system. The specifications are used to retrieve reusable ada components from a large software base. An automated transformation scheme then binds the retrieved components together based on the PSDL description. The prototype is then compiled and executed. The following sections describe each of the three major subsystems of CAPS.

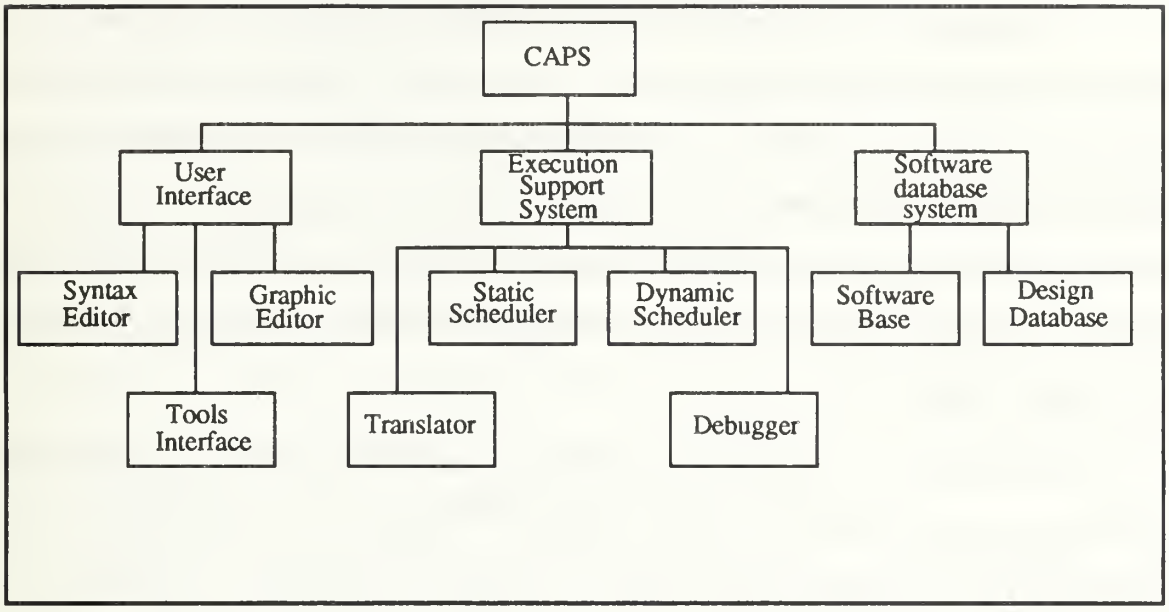

Figure 1.2: Tools in the Computer-Aided Prototyping System

1. User Interface

The user interface consists of a syntax directed editor, a graphic editor, and a tool interface. The graphic editor is used to create a graphic representation of the system in the form of a data flow diagram, plus timing and control constraints. Graphical objects used to represent the sytem include operators, inputs, outputs, data flows, and operator loops [13]. The syntax directed editor provides a convenient way of entering additional annotations to the graphics in the form of PSDL descriptions while preventing syntax errors. The tool interface hides the details of the interfaces of the CAPS tools from the designer [7].

2. Software Database

The software database system consists of an engineering design database system and a software database system [16]. The design database contains PSDL descriptions for all software projects developed using CAPS. The software base provides reusable software components for realizing given PSDL specifications [9]. The key to component storage and retrieval is the component's PSDL specification [18].

3. Execution Support System

The execution support system consists of four tools: translator, static scheduler, dynamic scheduler, and a debugger [7,16]. The translator generates codes that binds together the reusable components extracted from the software base. The static scheduler designates time slots for operators with real-time constraints before execution begins. The dynamic scheduler allocates time slots for operators that are not time critical. The debugger monitors timing constraints and design integrity as the prototype runs and reports failures.

C. OBJECTIVES

For a component to be added to the CAPS software base it must be specified in the prototyping system description language (PSDL). Writing these specfications is a time consuming process. One goal of this thesis is to design and implement a translation tool which takes an Ada specification as input and generates the PSDL interface required for storage in the CAPS software base. Another goal is to prepare and test Ada components for inclusion in the software base. These components must meet certain criteria to be usable in a larger system.

D. ORGANIZATION OF THESIS

An overview of the design of the CAPS software base and necessary requirements for the storage of components in the software base is provided in Chapter II. Chapter III discusses general characteristics required for component reuse, why Ada was chosen as the implementation language for the components, and the purpose of these components. A description of the design and implementation of the specification interface generator is provided in Chapter IV. Chapter V contains conclusions and recommendations for future research.

II. SOFTWARE REUSE

A. APPROACHS TO SOFTWARE REUSE

One of the major purposes of software reuse is to reduce the cost of software development and maintenance. Software reuse comes in many forms and occurs whenever artifacts or knowledge about the development of one system is reapplied in the development of another [6]. Examples include the reuse of code, designs, application generators, formal specifications, and off the shelf commerical packages. Collectively these examples are referred to as reusable software components.

A report in 1984 stated that "of all the code written in 1983, probably less than 15% is unique, novel, and specific to individual applications. The remaining 85% appears to be common, generic and concerned with putting applications onto computers" [5]. Thus, common generic software is an essential target of opportunity. There is almost no cost involved in copying a piece of software. Software reuse also provides a natural way to improve the quality of software. Frequent reuse of a software component can lead to frequent evaluation and revision, thus resulting in the construction of a high quality piece of software. Using high quality, well understood components as building blocks to construct large complex sytems should increase the quality of the final product and at the same time accelerate software production.

Technologies applicable to software reuse can be classified into two categories: reuse-in-the-small and reuse-in-the-large [6]. Reuse-in-the-small is concerned with the reuse of small pieces of source code such as classes, subroutines, Ada packages, and so on, and is the focus of this thesis. Reuse-in-the-large is concerned both with the reuse of large-grain components such as subsystems and the reuse of elements beyond source code such as design structures and decisions, domain knowledge, analysis information, and so forth.

Technologies applied to reuse-in-the-small are basically concerned with component representation and component management. Component representation deals with the form

and content of software components. There are some important characteristics specific to reusable components which will be discussed in Chapter III. Component management is concerned with classification, storage and retrieval of software components which will be reviewed in the following sections of this chapter.

B. RETRIEVING REUSABALE COMPONENTS

There are two costs associated with reuse-in-the-small. The first cost is associated with building and maintaing a component for reuse. The second is the cost associated with storing and retrieving components. The latter has resulted in an increasing demand for tools that aid in classifying, storing, and retrieving components. This section discusses some of these methods and the systems which use them [16].

1. Retrieval Methods

Most of the tools developed to assist in the retrieval of software components use one or more of three different approaches: browsers, informal specifications, or formal specifications. A brief description of each follows:

a. Browsers

A browser is a tool, usually window based, for looking through a collection of software components. The purpose of a browser is to allow the user to direct a search through the available components. This can be useful for a user who is familiar with the content and structure of a software collection. However, this method of retrieval is not suited for a very large software base. The user can easily miss semantically similiar components stored in separate areas of the software base. Also a user will not know when to stop looking for a component unless the component is found or the entire software base has been viewed.

b. Informal Specifications

Retrieval methods based on informal specifications require the user to list some attributes of the component sought. These attributes are used to direct the user to the

approriate components. Examples of this method include keyword search, mulit-attribute search, and natural language interface.

To perform a keyword search, a user specifies a list of words relevant to the component being sought. For example, a user looking for a component which implements a mathematical set, would list the keyword set. A major disadvantage to keyword search is that the choice of words listed is crucial to success. One keyword may lead to the retrieval of many inappropriate components that the user must review. On the other hand, the use of too many keywords may result in missing appropriate components.

A multi-attribute search is an extension of the keyword search. Attributes such as component class (procedure, function, package, etc.) or the number and types of parameters are used in the search. This type of search is generally more selective but requires the user to be familiar with the classification and storage techniques of the system.

A natural language search is based on a natural language query formed by the user. Although a user may find it easy to formulate a query using a natural language, this type of technique is very difficult to implement. Mechanisms built based on this method have been limited to certain domains or the use of a restricted language.

c. Formal Specifications

Retrieval using formal specification provides for a higher degree of automation. The user formulates a query using a high level language to specify the functionality of the desired component. Each component in the software base is stored with an interface using the same specification language. The system looks for components in the software base whose specification matches that of the user's query. However, writing formal specifications for components is difficult and requires substantial training.

2. Review of Current Systems

This section describes current retrieval systems that have been built and the methods used by each system.

a. Draco

The Draco project was developed at the University of California, Irvine and was one of the first systems to reuse components from all phases of the software lifecycle, including designs and analysis information. The system organizes software components by problem areas or domains. The retrieval scheme is based on a mulit-attribute search. A classification scheme, called faceted classification, is used to aid in organizing and retrieving components. Each component is described by using a set of attributes. The set of attributes is defined by the problem domain. The values associated with attributes are selected from a controlled vocabulary.

The system is conceptually simple to use and relatively easy to implement. However, classification is generally not suitable for unconstrained domains. Also, semantically similar components may be missed, especially if stored in different domains.

b. RAPID

RAPID (Reusable Ada Packages for Information System Development) is an ongoing project sponsored by the U.S. Army Information Systems Software Development Center in Washington. The system is designed to classify, store, and retrieve reusable Ada packages in the information systems domain. RAPID uses a faceted classification scheme similiar to Draco.

c. Operation Support System

The Operation Support System (OSS) is an ongoing project being developed by the Naval Ocean Systems Center. One goal of the project is to establish a Navy software library. Currently the components stored in the library are large command, control, and communications software subsytems. The system supports component retrieval using faceted classification, keywords, and a textual browser.

d. Common Ada Missile Packages (CAMP)

The Common Ada Missile Packages is an ongoing project sponsored by the Department of Defense to develope a software engineering system supported by a software library of reusable Ada components. The system is directed to software for missile sytems. One of the main components of the system is the Parts Engineering System (PES) Catalog. The catalog system provides a menu driven interface for storing, modifying, and retrieving components. Each component has an attribute list associated with it which is used as the basis for retrieval. The method of retrieval is based on muli-attribute search since one or more attributes may be used to drive a search.

e. CAPS

The method used in the Computer Aided Prototyping System is to retrieve components from a software base using a formal specification.The System also supports keyword searches and component browsing. The aim of CAPS is automated retrieval and integration of a component into a prototype based on formal specification. A description of the specification language PSDL is in section C and the basics of the retrieval sytsem will be discussed in section D of this chapter.

C. PROTOTYPE SYSTEM DESCRIPTION LANGUAGE (PSDL)

The prototype system description language (PSDL) and a large software base of reusable components form the basis of the CAPS system. PSDL is a specification language that was designed to support rapid prototyping of large real-time sytems [8,10]. PSDL contains a small set of powerful constructs which make it simple and easy to understand. The language was also designed for specifying retrieval of reusable modules from a software base.

PSDL is based on a computational model consisting of operators and data streams. A system is designed as a network of operators connected by data streams augmented with timing and control constraints. Operators can be either atomic or composite. A composite operator may be decomposed into a set of lower level operators and streams. An atomic

operator cannot be further decomposed. The network can be graphically represented as a set of data flow diagrams. The prototype as a whole is viewed as an operator which is the top level of the data flow diagram. The top level operator is decomposed into a set of more refined operators, and these are decomposed iteratively until all operators are atomic.

Operators can be either functions or state machines. The data streams can carry exception conditions or values of abstract data types [9]. A data stream which carries an instance of an abstract data type is defined as a PSDL type. This definition includes all of the operators that can operate on that data type. PSDL operators and types are the basic building blocks of a prototype.

A PSDL implementation of a prototype has two parts: a network consisting of the operators in the sytem and their interconnections, and a set of reusable components containing implementations of the atomic components in Ada. The Ada components are retrieved from the software base based on a PSDL description provided by the designer for each atomic component. The specification part of a PSDL component contains several attributes that describe the interface and behavior of that component. Figure 2.1 shows an example of a PSDL specification for an abstract data type for a set.

D. CAPS SOFTWARE BASE AND COMPONENT RETRIEVAL

The CAPS software base is an object-oriented database which contains PSDL descriptions and code for all available reusable software components. The database management system supports automatic retrieval and provides graphical tools for browsing and doing keyword searches [14]. Graphical tools provide a means for storing components in the software base as well.

```
type SET specification

   operator EMPTY specification
      output S1 : set end

   operator ADD specification
      input ELEMENT : integer, S1 : set
      output S2 : set end

   operator IN specification
      input ELEMENT : integer, S1 : set
      output RESULT : boolean end

   operator SUBSET specification
      input S1, S2 : set
      output RESULT : boolean end

   operator EQUAL specification
      input S1, S2 : set
      output RESULT : boolean end

   keywords SET, INTEGER
   description {Implements a set of integers}
   axioms
   {obj SET is sort Set.
      protecting INT.
      op empty : -> Set.
      op add : Int Set -> Set.
      op in : Int Set -> Bool.
      op subset : Set Set -> Bool.
      op equal : Set Set -> Bool.
      vars s1 s2 : Set.
      vars e1 e2 : Int.
      eq in (e1,empty) = false.
      eq in (e1,add(e2,s1)) = or (= =(e1,e2), in(e1,s1)).
      eq subset(empty,s1) = true.
      eq subset(add(e1,s1),s2) = and(in(e1,s2),subset(s1,s2)).
      eq equal(s1,s2) = and (subset(s1,s2),subset(s2,s1)).
   endo}
end
```

Figure 2.1: A PSDL Specification for a Set

To store a component in the software base requires three files. A PSDL specification, and the interface and body for the implementation code [17]. The PSDL specification gives a means to uniformly specify the functionality of the component as described in the interface code. The PSDL specification is passed through syntactic and semantic normalization before being stored in the software base (see Figure 2.2). The normalization process modifies the specification to improve the efficiency of the search. A query for a

library component is formed by constructing the PSDL specification for the desired component. The query specification is normalized then matched against the stored specifications (see Figure 2.3).

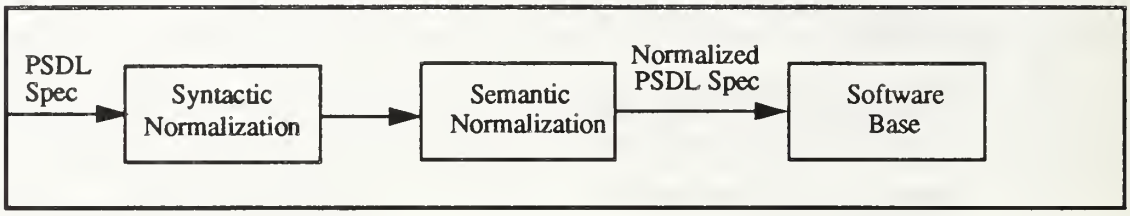

Figure 2.2: Component Storage

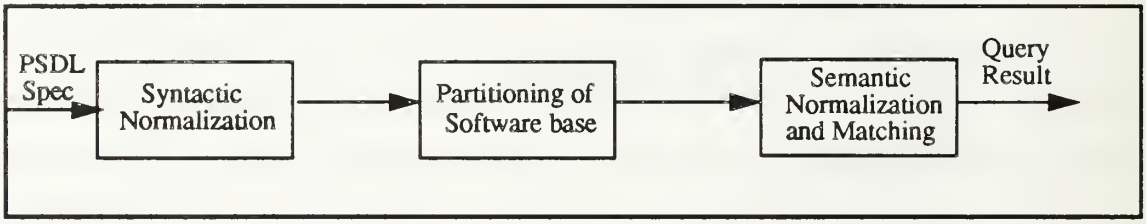

Figure 2.3: Component Retrieval

The retrieval process is two phased [11,17]. Syntactic matching takes place first and partitions the software base quickly in order to narrow the list of possible candidates that will be considered during semantic matching. The syntactic phase uses the number and types of parameters associated with each component to narrow down the search. The semantic matching phase uses the axioms in the latter half of a PSDL specification (see Figure 2.1) to narrow the set of candidates further. This phase determines which components are behaviorally close to the query.

The ability to accurately specify reusable components with PSDL is critical to the success of automated retrieval. The CAPS software base was designed to store components implemented in various programming languages. Because of the differences in the capabilities of different programming languages, the software base is separated into language domains. PSDL is not geared toward any particular programming language and therefore must be refined to specify a particular programming language. The software base

is designed to recognize the enumeration of PSDL for any language in the following areas: generic parameters, abstract data types, type inheritance, and array abstract data type [14]. For a particular language library the definitions of the special identifiers are contained in a rule file for each library. The rule file established for the Ada library guided the translation of Ada specs into a PSDL interface (see Chapter IV).

III. REUSABLE SOFTWARE COMPONENTS

A. ENGINEERING SOFTWARE COMPONENTS TO FACILITATE REUSE

To fully derive the benefits of reusable software components, we must engineer our software with reuse in mind and begin to accumulate a rich set of components for the designer [1]. There are three factors which make it practical to formalize reuse-in-the-small: the emergence of a widely accepted body of knowledge about data structures and algorithms, development of software engineering principles, and the development of programming languages, such as Ada, which support reuse. Components built for reuse should exhibit the characteristcs of any good piece of software. For example, a component should be maintainable, efficient, reliable, understandable, and, of course, correct.

A reusable component should be cohesive and loosely coupled. Cohesive means the component should denote a single abstraction. Loosely coupled means the component should be defined independently of other abstractions. For example, a component which denotes an abstraction for real numbers should not include an abstraction for a set. Secondly, the abstraction should not be dependent on other abstractions.

There are other desirable characteristics of reusable components. They should be sufficient, complete, and primitive [1]. These characteristics of a component have to do with the outside view of the component. A sufficient component captures enough characteristics of the abstraction to permit meaningful interaction with the object. For example, if a component represents real numbers but provides no means for adding two real numbers together, then the component is useless. The interface of a complete component captures all characteristics of the component. Sufficiency implies a minimal collection of operations, whereas a complete set of operations is one that covers all aspects of the underlying abstraction.

Primitive operations are those that can be effiiciently implemented with only access to the underlying representation. For example, the addition of two real numbers. Those operations which are not primitive but may be useful to the component can be easily

extended by the user of the component by building new composite operations. For example, the abstraction of real numbers includes the notion of comparing two numbers. To make the component sufficient it should include "<"and "=" operations. To enhance the completness of the abstraction, we might add a ">" operation. However, a ">" can be built as a composite operation of the "<" and "=" operations. The designer of a reusable component cannot know exactly how a particular component is going to be used.

B. WHY ADA?

Ada is a language that embodies and enforces modern software engineering principles. Ada was also designed with the explicit requirement to support reuse. Features of Ada which support reuse include [1,2]:

1. Separation of interface from the body,
2. Generic program units,
3. Strong typing, and
4. Variety of program units including packages and subprograms

Separation of the interface from the body supports information hiding and abstraction. An ada specification (interface) identifies the functionality of a component and is the information visable to the user of that program unit. The body contains the unit implementation which is hidden from the user. Unimportant details are hidden from the user. Separation of the interface from the body is also important for storing and classifying a component.

The Ada generic unit is the main mechanism for building reusable components. A generic unit provides the template for the algorithm or data structure. Upon instantiation of a generic unit the client provides the set of allowable values for a data structure.

Ada is a strongly typed language. This means objects declared of a given type may only take on those values which are legal for that type. In addition, the only operations that may be carried out on an object are those which are defined for its type. Strong typing can be instrumental in improving the reliability of a program unit by guaranteeing that the

properties of an object are not violated. The requirement for explicit declaration of objects with their types improves the readability of a program. This guarantees the programmer must say something about the property of an object.

Packages permit the encapsulation of a group of logically related entities which directly supports data abstraction and information hiding. Well structured Ada systems are decomposed into levels of abstractions, structured as collections of logically related packages that form a model of reality.

C. ADA COMPONENTS SELECTED FOR SOFTWARE BASE

The components selected and tested as a part of this study for inclusion in the CAPS software base all represent an abstract data type. They include an unbounded map, an unbounded set, a bounded multiset, a bounded graph, an unbounded graph, a matrix, vector, real numbers, and bounded integer types. All components were developed by students at the Naval Postgraduate School. The author of this thesis tested and in most cases modified the components. Operations were added to components which did not sufficiently define the abstraction. Other changes corrected errors and improved efficiency and readability. Appendix A contains the specification for each component. The following is a brief description of the abstraction, implementation, and major changes made to the components.

1. Unbounded_Set

a. The abstraction

The mathematical abstraction set is widely used in computer science applications. Many interesting data structures can be thought of as just implementations of sets. Thus it makes sense to build a reuseable set component. Given a "universe" of permissable values, a set is an unordered collection of objects belonging to that universe. Two sets are said to be equal if they have the same members. A set is said to be empty if it has no members. What are the important operators associated with sets? Certainly adding and removing an element from a set are essential as well as being able to determine the size

of the set. Other important operations include the dyadic operations union, intersection, difference, and equality.

b. Implementation

The Unbounded_Set was implemented with a linked list structure. The number of elements that can be added to the set are only limited by available memory. The unit contains one generic type parameter which defines the type of elements which may be added to the set. A hashing function could be used to simplify searching for a given element, however, it would complicate the algorithms for the dyadic operations.

c. Major changes

A remove procedure and a set difference procedure were added to the original version to make the component sufficient. An error in the original add procedure which allowed duplicate copies of an element was corrected.

2. Bounded_Multiset

a. The abstraction

A multiset is the same as a set except that items contained within a mulitiset need not be unique. The size of a mulitset refers to the number of items contained within the set. A multiset which contains no items is said to be empty. Operations important for a multiset include an empty set constant, a count operation that returns the number of instances of a given element in a given multiset, add and remove operation, an operation to test equality, and a size operation which returns the number of distinct elements in a multiset.

b. Implementation

The bounded_multiset is implemented with an array. The unit has two generic parameters. Element is a generic type parameter which provides the type of items which can be added to the set. Max_Size is a generic value parameter which determines the size of the array (the number of unique elements that may be added to the set).

c. Major Changes

The original data structure kept multiple copies of any repeating elements in the array. The data structure was changed to keep a count of each element rather than keeping mulitple copies of the element. This representation is more efficient for any application which contains many copies of an element, for example an inventory record. Two procedures, Last_Element and First_Element, were removed from the package specification and are now local to the body. They were used in the implementation of the data type, but are not part of the definition of the data type, hence should not be visable to the user. An overflow exception was added to the add procedure. The function empty_set was changed from a predicate function to an empty set constant. Get and put operations were removed from the test package and added to the generic package.

3. Bounded_Graph

a. The abstraction

Graphs are an important mathematical structure and are used widely in computing problems. A graph is made up of a set of vertices or nodes and a set of arcs or edges, which represent connections between the vertices. Our abstraction represents a directed graph or a graph in which the edges have direction. Important operations include initializing a graph to empty, adding a node, adding an edge, checking whether there is an edge between a given pair of nodes, finding the set of nodes connected to a given node via an outgoing or incoming edge, removing the edge between a given pair of nodes, and removing a node and all of the edges connected to that node.

b. Implementation

The bounded_graph is implemented using an adjacency matrix. An adjacency matrix is a $n \times n$ boolean matrix where n represents the maximum number of nodes allowed in the graph. If the $[i,j]$ element in the matrix is true then there is an edge from vertix i to vertix j and false if there is not. A one dimensional array of size n is used to store the values

values of each node. The unit has two generic parameters. One is a type parameter which allows the user to define the allowable values for the nodes. The other is a value parameter of type positive and determines the maximum size of the graph.

c. Major Changes

The original version was a machine with a state variable representing the graph declared in the specification. The state variable was removed and a parameter representing a graph type was added to all functions and procedures. An operator which returns an empty graph was added and overflow exception was added to a local procedure which is called by the add operation.

4. Unbounded_Graph

a. The abstraction

The abstraction for the unbounded graph is the same as the bounded graph, but allows an arbitrary number of nodes to be added to the graph.

b. Implementation

The unbounded graph permits an arbitrary number of nodes in the graph. The implementation uses an adjacency list. The basic idea of an adjacency list is to list each vertex followed by the vertices adjacent to it. This provides the basic information about a graph: the vertices and edges. Two linked lists are used. One list links all the nodes in the graph. Each node in the graph has an adjacency list, which lists all adjacent nodes. The unit has one generic type parameter which imports the allowed values for each node.

c. Major Changes

The original data structure was modified to improve readability. An adjacency node in the original structure used a pointer to a graph node to identify an adjacent vertix. This pointer was removed and replaced with the node element. Type declarations used to build the graph data type were moved from the public to the private

section of the specification. This hides unnecessary implementation details from the user. An unnecessary type declaration was removed to improve readability.

5. Unbounded_Map

a. The abstraction

A map permits one to define arbitrary relationships among otherwise unrelated objects. A map is a mathematical function on objects of one type, called the domain, yeilding objects of another type called the range [1]. Thus a map consists of a dynamic collection of bindings from the domain to the range. Bindings may be added, removed, and modified over the lifetime of a map. The extent of a map represents how many bindings are in the map and if a map contains no bindings the map is said to be empty.

Operations include initializing a map to empty, adding a binding to a map, finding the range value associated with a given domain value, checking whether a given domain value has a binding in the table, finding the number of bindings in the table, and removing the binding associated with a value of the domain type.

b. Implementation

An unbounded abstraction should permit a map with an arbitrary number of domain and range pairs. This can be easily done using a linked list structure whose nodes are records containing these pairs. However, to mitigate the time for searching a gigantic list for a given pair, the map is represented as a collection of several smaller lists. The unbounded map is represented as a set of blocks of lists. Each block is an array where each array component acts as a bucket which holds a list of ordered pairs. A set of blocks is represented using a linked list. A hashing function is used to determine in which list or bucket a map pair will be located. If the hash function returns an index outside the available map blocks, a new block is added.

The generic unit has eight generic parameters. Two type parameters are used to import the domain and range types. A value parameter allows the user to determine the number of buckets per block. The remaining parameters are generic subprogram

parameters. One is a hashing function, the others are get and put procedures for the domain and range.

c. Major Changes

In the original version, two state variables, NUM_BINDINGS and NUM_BLOCKS, were declared in the package body. These variables will reflect inaccurate data if more than one instance of the map type is declared. Thus they were removed from the body and made part of the data structure of the map type.

6. Real_Numbers

a. The abstraction

The real abstraction is a high precision real number type representing the standard mathematical domain of real numbers. Real literals are decimals, with at least one digit on each side of the decimal. Operations include conversions from Ada float to real, addition, subtraction, multiplication, division, and the comparison operators "=" and "<".

b. Implementation

The real number data type is implemented as a record type. The record contains three fields: the sign of the number, a digit array containing the digits of the real number, and an exponent array. The unit has two generic value parameters, digits and max_exponent. Digits represents the minimum required precision and determines the size of the digit array. Max_exponent determines the largest number which can be represented. The representation will handle numbers ranging up to 10^e where e is the max_exponent.

c. Major Changes

The original version used separate arrays to hold the whole and decimal parts of a real number. The whole array was declared as a non-generic static type. This version was simplified by removing the static array and normalizing the position of the decimal. This allowed the elimination of five local subprograms and the simplification of others.

Type declarations used to build the real data type were moved from the public to the private section of the specification. This eliminates unnecessary implementation details from the user.

7. Bounded_Integer

a. The abstraction

The bounded integer abstraction represents signed whole numbers of standard mathematics. The range of the integers is bounded. Operations include conversion from Ada integers to a bounded_integer, addition, subtraction, multiplication, division, mod, and the comparison operators "=" and "<".

b. Implementation

The bounded_integer type is implemented using an array. The unit has one generic value parameter which specifies the number of decimal digits the representation must support.

c. Major Changes

Type declarations used to build integer type were moved from the public to private section of specification. Many algorithms were rewritten to improve readability. This mainly involved the removal of unnecessary local variables.

8. Vectors

a. The abstraction

A mathematical vector is a set of elements which is ordered in the sense that each component is assigned a specific position in the set. The dimension of a vector designates the number of elements for a given vector. Operations include conversion from an array of elements to a vector, vector addition and subtraction, multiplication by a scalar (value of the element type), dot product of two vectors, and the mathematical length of a vector.

b. Implementation

The vector is implemented using a one dimensional array. The generic unit has a generic type parameter which determines the element type. A generic value parameter is used to import the dimension of the vector. Generic subprogram parameters are needed to define arithmetic operations on the element type.

c. Major Changes

An operation to convert an array of objects to a vector was added.

9. Matrix

a. The abstraction

A mathematical matrix can be viewed as a rectangular array of elements, having R rows and C columns. Any particular element in the matrix may be referred to using to subscripts, the row and column position. Operations on matrices include conversion from an array of elements to a matrix, matrix addition and subtraction, multiplication by a scalar (value of the element type), matrix multiplication, and transposition.

b. Implementation

The matrix type is implemented using a dimensional array. Two generic value parameters are used to define row and column length. A generic type parameter defines the element type.

c. Major Changes

None.

IV. PSDL INTERFACE GENERATOR

The PSDL interface specification generator is a tool which automates the process of producing the necessary PSDL interface for the storage of an Ada component in the CAPS software base. This chapter describes the design and implementation of the translator. Section A provides information, extracted from [4], that is necessary to understand the code listed in Appendix B.

A. KODIAK

Kodiak is the tool we used to build the translator. It is a fourth generation language developed at the University of Minnesota and designed for the purpose of producing language translators. The language is based on Knuth's description of attribute grammars [4]. Attribute grammars are a scheme for describing syntax-directed translation, in which a context-free grammar's rules are augmented with equations defining these attributes. The string is parsed into a syntax tree by applying a set of grammar rules. The root of the tree represents the start symbol, the leaf nodes represent the terminal symbols, and the internal nodes represent the non-terminal symbols of the grammar. Attributes can be assigned to the nodes of the syntax tree. To translate the input string into an output string, values are assigned to the attributes of each node. The root of the tree is given an attribute whose value is based on the collective value of all the nodes in the tree, producing the output string.

The values of a particular node's attributes can be determined in one of two ways. The values of the attributes are either inherited or synthesized. Synthesized attributes are evaluated from the bottom up, meaning the value of an attribute at a given node is derived from that node's descendants. Inherited attributes are evaluated from top down, meaning the value of an attribute at a given node is derived from the node's parent.

Every Kodiak program consists of three sections, as shown in Figure 4.1. The first section describes the features of the lexical scanner which is used to translate the source text into tokens and associativities for those tokens. The second section declares the attributes and their type associated with each grammar symbol. The third section describes

the grammar and attribute equations which define the semantics of the translation. Each section must be separated by a double per-cent symbol (%%) on a line by itself . The symbol "!" introduces a comment and everything written after it on a line it will be interpreted as a comment. The following is a brief description of each section.

```
! Lexical Scanner Section


%%
! Attribute Declaration Section


%%
! Grammar rules and attribute equations
```

Figure 4.1: Kodiak Program Structure

1. Lexical Scanner

The lexical scanner section of a program defines the terminal symbols (leaf nodes of the syntax tree) of the source language and how these symbols are to be transformed into tokens. The source language in this case is Ada and the terminal symbols correspond to Ada's lexical units. Ada's lexical units consist of identifiers, numeric literals, character literals, and comments. Ada's reserved words, a subset of identifiers, must be defined in this section as they are required in defining the grammar. The basic form of a token definition is:

TERMINAL_NAME : REGULAR_EXPRESSION

Terminal name is the name given to the token and appears in the definition of the grammar in section three of the program. The regular_expression provides the definition of the token. Code fragments from the lexical scanner section of the PSDL interface generator are given in Figure 4.2. These declarations indicate that an occurence of the regular expression to the left of a colon is to be replaced by the atomic terminal symbol on the right. For example, an occurence of the string "package" or "PACKAGE" in the input text is to be replaced by

the symbol PACKAGE. The keyword %define introduces a definition. Square brackets enclosing a set of characters indicates that any character of the set may be used to match the text. Curly braces are used to invoke a substitution. The '+' operator indicates that one or more of the previous class may be used to match the text. The '*' operator indicates that zero or more of the previous class may be used to match the text.

```
!definition of lexical classes

%define Digit : [0-9]
%define Int : {Digit}+

!definition of compound symbols and keywords

PACKAGE : package | PACKAGE
REAL_LITERAL : {Int}"."{Int}
```

Figure 4.2: Examples of Token declarations

2. Attribute Declarations

The attribute declarations section consists of attribute declarations for all non-terminals and terminals named in the program. Kodiak supports two primitive types for attributes: strings and integers. Kodiak also supports higher order map types. For example in Figure 4.3, generic_type_definitions has two attributes. The attribute variable_type is declared to be of type string. The attribute generic_types is declared to be a map type whose domain and range are both of type string. Basic_declaration has one attribute, number_of_operators, which is declared to be of type integer.

Terminal symbols may also have attributes. These symbols are permitted two predefined attributes called %text and %line in addition to user defined attributes. In Figure 4.3, the terminal IDENTIFIER has the attribute %text which will be initialized to the text the terminal symbol IDENTIFIER matched in the input text.

```
!Attribute declarations for non-terminal symbols

start { psdl_interface_specification : string; };

generic _type_definition { variable_type : string;
                           generic_types : string->string; };

basic_declaration { number_of_operators : int; };

!Attribute declarations for terminal symbols

IDENTIFIER {%text : string; };
```

Figure 4.3: Example of Attribute Declarations

3. The Attribute Grammar and Equations

The attribute grammar section consists of a set of BNF rules defining the grammar of the source language. Each rule is associated with a set of equations which define how the input text is to be translated. A fragment of the PSDL interface generator is given in Figure 4.4. The fragment defines the grammar rule for the non-terminal package_specification. The symbol "|" is used to separate two definitions for the grammar symbol. Curly braces surround any attribute equations. Null productions are permitted, meaning the curly braces may be left empty.

An attribute is referred to using dot notation. The grammar symbol associated with the attribute proceeds the dot and the name of the attribute follows. If more than one occurence of a grammar symbol appears in a grammar rule, the leftmost symbol is taken to be the one referred to by an attribute. To refer to a later symbol, the attribute may be followed by a number in brackets referring to the symbols position of occurence in the rule.

Kodiak supports traditional arithmetic and relational operators for integers and strings. An addition operator for map types is provided using the symbol "+|". A conditional clause is also supported and is exemplified in Figure 4.4. The then portion of the clause follows the arrow,"->", and the else portion follows a "#" sign.

One attribute equation is defined between the first set of curly braces in Figure 4.4. The equation evaluates the value for the attribute psdl_interface_specification. This equation contains a conditional. If the attribute number_of_operators equals one, then the value following the arrow is assigned to the psdl_interface_specification, otherwise the value following the # sign is used. Literal strings appear between quotes. The other attributes which appear in the equation are evaluated to strings. All literals and attributes between the brackets are catenated to produce the psdl_interface_specification. A further explantion of the attributes in provided in section C.

```
package_specification
      : PACKAGE IDENTIFIER IS basic_declarative_items END IDENTIFIER ';'
         { package_specification.psdl_interface_specification =
            basic_declarative_items.number_of_operators == 1
              -> [ "OPERATOR", IDENTIFIER.%text, "\nSPECIFICATION\n",
                    package_specification.generic_type_declarations, "\n",
                    basic_declarative_items.input_parameters,
                    basic_declarative_items.output_parameters,
                    basic_declarative_items.exceptions,
                  "END\n"]
              # [ "TYPE ", IDENTIFIER.%text, "\nSPECIFICATION\n",
                    package_specification.generic_type_declarations, "\n",
                    basic_declarative_items.type_declarations,
                    basic_declarative_items.operator_specifications,
                  "END\n"];
      | PACKAGE IDENTIFIER IS basic_declarative_items PRIVATE
                                  basic_declarative_items END IDENTIFIER ';'
         { package_specification.psdl_interface_specification =
            basic_declarative_items.number_of_operators == 1
              -> [ "OPERATOR ", IDENTIFIER.%text, "\nSPECIFICATION\n",
                    package_specification.generic_types_declarations, "\n",
                    basic_declarative_items.input_parameters,
                    basic_declarative_items.output_parameters,
                    basic_declarative_items.exceptions,
                  "END\n" ];
              # [ "TYPE ", IDENTIFIER.%text, "\nSPECIFICATION\n",
                    package_specification.generic_type_declarations, "\n",
                    basic_declarative_items.type_declarations,
                    basic_declarative_items.operator_specifications,
                  "END\n" ]; }
;
```

Figure 4.4: Example of Attribute grammar and equations

B. MAPPING ADA TO PSDL

An Ada specification will be translated into either a PSDL operator or PSDL type depending on the number of procedures and functions declared in the specification. If only one procedure or function is declared, the specification is translated as a PSDL operator using the template in Figure 4.5. The operator will be given the name of the procedure or function name in the Ada specification. If the specification contains 0 or 2 or more procedures or functions, the specification is mapped to a PSDL type using the template in Figure 4.6 . The type will be given the name of the Ada package.

```
OPERATOR
SPECIFICATION
   GENERIC
      generic_parameter_declarations
   input      input_parameters
   output     output_parameters
   exception_declarations
END
```

Figure 4.5: Operator Template

```
TYPE
SPECIFICATION
  GENERIC
     generic_parameter_declarations
  type_declarations
  operator_specifications
END
```

Figure 4.6: Type Template

variable_names : GENERIC_VALUE

Figure 4.7: Example of a generic_parameter_declarations Template

Each generic declaration in the Ada specification is translated into a single PSDL generic type or built into a composite generic type. Generic value and object parameters declared in the Ada specification are translated into a single PSDL generic type. The parameter name is the same as the parameter in the Ada specification and the type is translated as GENERIC_VALUE. Generic type parameters declared in the Ada specification are also translated into a single PSDL generic type if they are not part of a generic array type definition. The parameter name is the same as the parameter in the Ada specification and the type is translated as GENERIC_TYPE. If the index and element part of an array type parameter are generic parameters, they are incorporated into the definition of the array type of the PSDL. For example given the following Ada declaration:

```
generic
   type ELEMENT is (<>);
   type LIST is array (INTEGER range <>) of ELEMENT;
```

only one PSDL generic type will be translated from the two generic Ada parameters as follows:

```
LIST : GENERIC_TYPE [ BASE_TYPE : ARRAY [
                ELEMENT : DISCRETE, INDEX : INTEGER ]]
```

Generic subprogram parameters declared in the Ada specification are translated into a single generic PSDL type . The parameter name is the same as the name given in the Ada specification, with the exception of overloaded functions named with symbols. Operator symbols are translated into strings. For example, a generic subprogram named "+" will be renamed to "add". The type is translated as GENERIC_PROCEDURE.

C. EXPLANATION OF ATTRIBUTES

The following is a brief description of the attributes used to build the PSDL interface specification generator.

1. psdl_interface_specification

The attribute psdl_interface_specification is the highest attribute and stores the result of the translation. The translation is written to a file named with the Ada package name concatenated with the suffix ".psdl".

2. file_name

The attribute file_name is a synthesized attribute which provides the name of the file the translation is written to.

3. generic_type_declarations

The attribute generic_type_declarations is a synthesized string which builds the generic portion of the PSDL specification.

4. number_of_operators

The attribute number_of_operators is a synthesized integer used to determine if the PSDL specification is that of an operator or a type. This attribute counts the number of procedures and functions declared in the Ada specification. If only one procedure or function is declared, the operator template, Figure 4.5, is used to build the specification, otherwise the type template, Figure 4.6, is used to build the specification.

5. operator_specification

The attribute operator_specification is a synthesized string which builds the operator specifications for a type declaration.

6. type_declarations

The attribute type_declarations is used to build the non-generic type declarations of the PSDL specification. In this implementation only private type declarations are translated. They are translated to the type ADT.

7. input_parameters

The input_parameters attribute is used to build the input parameters for each operator in the PSDL specification. Ada in and in out variables of procedures and functions become the input parameters for a PSDL operator. The name and the type name of a PSDL input will be that of the corresponding Ada parameter.

8. output_parameters

The output_parameters attribute is used to build the output parameters for each operator in the PSDL specification. The name and the type name of a PSDL output will be that of the corresponding Ada parameter.

9. exceptions

The attribute exceptions provides the exception declarations for a PSDL operator interface. Type interfaces do not have exceptions declarations included with the operators in this implementation.

10. variable_type

The attribute variable_type is a synthesized string which provides the type name for variables declared in the generic portion and type declaration of the PSDL specification.

11. variable_name

The attribute variable_name is a synthesized string which provides the name for each input/output parameter of an operator.

12. mode

The attribute mode is a synthesized map used to determine if there are any input or output parameters to an operator specification. It is used to modify the template in Figures 4.5 and 4.6. If there are no input parameters, the fixed input portion of the template is to be eliminated. If there are no output parameters, the output portion is eliminated.

13. mode_check

The attribute mode_check is an inherited map which is used to initialize the attribute mode to a default value of empty string.

14. current_mode

The attribute current_mode is a synthesized string used to determine if a comma is required between two parameters.

15. composite_types

The attribute composite_types is an inherited map which is used to determine if a generic type declaration in the ada specification is the index or element type of an array declaration. This information is used to build an array declaration in the PSDL.

16. new_composite_types

The attribute new_composite_types is a synthesized map which is built to provide the value of the map composite_types.

17. generic_types

The attribute generic_types is an inherited map which provides the type name for the index and element part of an array declartion.

18. new_generic_types

The attribute new_generic_types is a synthesized map which is built to provide the values of the map generic_types.

19. comma

The attribute comma is a synthesized string used to determine if a comma is needed between two generic parameters.

D. SAMPLE INPUT AND OUTPUT

Two samples of Ada specifications used as input and the respective PSDL interface specifications generated are shown below. The first example is translated into a PSDL operator, the second into a PSDL type.

1. Operator Example

The first example in Figure 4.8 shows an Ada specification for a generic package which contains one subprogram and four generic parameters. The generated PSDL specification is shown in Figure 4.9. The PSDL specification is an operator and contains two generic parameters. The operator specification was a result of only one subprogram being declared in the Ada package. The three generic parameters in the Ada generated one type definition in the PSDL. This is because the types ITEM and INDEX are used to define the array ITEMS.

```
generic
   type ITEM is private;
   type INDEX is (<>);
   type ITEMS is array (INDEX range <> ) of ITEM;
   with function "<" (Left : in ITEM; Right : in ITEM) return BOOLEAN;

package Bubble_Sort is
    procedure Sort (The_Items : in out ITEMS);
end Bubble_Sort;
```

Figure 4.8: Ada Specification for generic bubble sort package

```
OPERATOR Bubble_Sort
SPECIFICATION
  GENERIC
     ITEMS : GENERIC_TYPE [ BASE_TYPE: ARRAY[ARRAY_ELEMENT : PRIVATE,
                                  ARRAY_INDEX : DISCRETE ] ],
     less_than : GENERIC_PROCEDURE

   input The_Items: ITEMS
   output The_Items: ITEMS
END
```

Figure 4.9: PSDL output for Ada generic bubble sort package

```
with text_io; use text_io;

generic
   type t is private;
   block_size : in natural := 128;
   with procedure eq (x,y : in t; v: boolean);

package sb_set_pkg is
   type set is private;
   type index_array is array (natural range <>) of natural;
   procedure empty (s: out set);
   procedure add (x: in t; si: in set; so: out set);
   procedure remove (x: in out t; s: in out set);
   procedure member (x: in t; s: in set; b: boolean);
   procedure union (s1,s2 : in set; s3 : out set);
   procedure difference (s1,s2 : in set; s3 : out set);
   precedure intersection (s1,s2: in set; s3: out set);
   procedure size (s: in set; v: out natural);
   procedure equal (s1,s2 : in set; v: out boolean);
   procedure subset (s1,s2: in set; v: out boolean);

   private

   type link is access set;
   type elements_type is array (1..block_size) of t;
   type set is
     record
       size : natural := 0; --The size of the set
       elements : elements_type; --The actual elements of the set
       next : link := null; --The next node in the list
     end record;

   --Elements (1..min(size,block_size)) contains data
 end sb_set_pkg;
```

Figure 4.10: Ada Specifiation for generic set

2. Type Example

The second example in figure 4.10 shows an ada specification for an abstract data type the "set". The generated PSDL specification is shown in Figure 4.11. Because there is more than one subprogram declared in the Ada, the resulting PSDL specifcation is for a type. The example shows the translation of generic type, value, and subprogam parameters. The private type declaration, set, translated to ADT in PSDL. All subprograms were translated to operators.

```
TYPE sb_set_pkg
SPECIFICATION
    GENERIC
        t: GENERIC_TYPE,
        block_size: GENERIC_VALUE,
        eq: GENERIC_PROCEDURE

    set : ADT

    OPERATOR empty
    SPECIFICATION
        output s: set
    END

    OPERATOR add
    SPECIFICATION
        input x: t, si: set
        output so: set
    END

    OPERATOR remove
    SPECIFICATION
        input x: t, s: set
    output x: t, s: set
    END

    OPERATOR member
    SPECIFICATION
        input x: t, s: set, b:

    OPERATOR union
    SPECIFICATION
        input s1, s2: set
        output s3: set
    END
    OPERATOR difference
    SPECIFICATION
        input s1, s2: set
        output s3: set
    END
    OPERATOR intersection
    SPECIFICATION
        input s1, s2: set
        output s3: set
    END

    OPERATOR size
    SPECIFICATION
        input s: set
        output v: natural
    END

    OPERATOR equal
    SPECIFICATION
        input s1, s2: set
        output v: boolean
    END

    OPERATOR subset
    SPECIFICATION
        input s1, s2: set
        output v: boolean
    END

END
```

Figure 4.11: PSDL output for Ada generic set package

E. LIMITATIONS

Ada is not case sensitive whereas Kodiak is. This presents a problem in parsing legal expressions in Ada. For example, the terminal symbol PACKAGE was defined to match any occurence of "package" or "PACKAGE" in the input string. However, it is legal in Ada to use a mixture of upper and lower case. This will not be selected as a match in this implementation. The way around this is to run all Ada specifications through a pretty printer first.

This implementation does not include the grammar for the entire Ada language. The grammar selected includes Ada package specifications and generic specifications. This implemention does not include keywords, descriptions, exceptions for type declarations, and OBJ3 axioms. Keywords and descriptions are not used for query by specification, but are required for keyword search and browser, respectively.

V. CONCLUSIONS

The use of reusable software components will be crucial to the successful development of large software systems. One of the major problems in code reuse is the lack of a large library of reusable software components. This problem is aggravated by the fact that more effort is required to build a generalized component for reuse then to build one for a specific application. A reusable component must be sufficiently powerful to accomodate a wide range of applications. A second problem with code reuse is concerned with the storage and retrieval of reusable components. Specifically, to automate component retrieval, it is necessary to accurately specify the component's functionality. The CAPS system is designed to exploit code reuse for rapid prototyping of hard-real time systems. The prototype is built in part by the automated retrieval of reusable Ada components.

A. ACCOMPLISHMENTS

As a part of this thesis, reusable Ada components were selected and prepared for inclusion in the CAPS software base. However, these components represent a small subset of the thousands of components which will be required. Another way to build up the number of components in the CAPS software base is to adopt components which are already being used in other software libraries. To store these components in the CAPS software base they will need to be specified in PSDL. It is important to accurately specify a component's functionality in PSDL since the key to successful retrieval is this specification. We have developed a translation tool that is able to generate in part the PSDL specification for Ada components.

B. FUTURE WORK

There is still a tremendous amount of work which needs to be done in this very labor intensive area of building reusable components. Depasquale [3] addresses the issue of automating the production of test programs based on a component's formal specification. More work could be done in this area to aid in component testing. If a component is built

based on a formal specification then it should be easier to test the component and at least part of the Ada code may be automatically generated (see Reference 15).

The translator should be expanded to include at least some of the limitations addressed in Chapter IV.

The opportunity and benefits for reuse are real. Building systems from reusable components should result in higher quality and more reliable systems. CAPS is one example of a system realizing these benefits.

APPENDIX A. ADA SPECIFICATIONS FOR REUSABLE COMPONENTS

A. UNBOUNDED_SET

```
--*****************************************************************************
--* Title : CS-4530 PROJECT 1 " U N B O U N D E D S E T "                      *
--* Author : Erhan SARIDOGAN                                                  *
--* Modifications : Procedure add modified March 92 by J.M. Sealander to prevent replicate *
--*elements being added to a set., procedure remove and function difference added to pkg. *
--* Date : November,8,1991                                                    *
--* Course : CS - 4530                                                        *
--* System : Unix                                                             *
--* Compiler : Verdixada                                                      *
--* Description : This generic package provides to create and manipulate      *
--* unbounded mathematical set of a given type.                               *
--* It has two generic parameters,Element_Type and function                   *
--* Equal.The I/O procedures need to use generic procedures.                  *
--* Type Unbounded_Set is declared as private type.                           *
--* All the required operations are available in the package                  *
--* Link list structure is used to provide unlimited entry.                   *
--*****************************************************************************

with TEXT_IO;
use TEXT_IO;

generic

    type ELEMENT_TYPE is private;
    with function EQUAL( X, Y : ELEMENT_TYPE ) return BOOLEAN is "=";

package UNBOUNDED_SET_PKG is

    type UNBOUNDED_SET is private;

    -- This array is needed to initialize a set variable with given values
    type SET_ARRAY is array (NATURAL range <>) of ELEMENT_TYPE;

    INVALID_SET_ENTRY,
    REPEATED_ELEMENT : exception; -- used in I/O

    -- Operations on sets ( The required ones )
    function EMPTY return UNBOUNDED_SET;

    procedure ADD( X : in ELEMENT_TYPE; S : in out UNBOUNDED_SET );

    procedure REMOVE( X : in ELEMENT_TYPE; S : in out UNBOUNDED_SET );

    function MEMBER( X : ELEMENT_TYPE; S : UNBOUNDED_SET ) return BOOLEAN;

    function UNION( S1, S2 : UNBOUNDED_SET ) return UNBOUNDED_SET;

    function INTERSECTION( S1, S2 : UNBOUNDED_SET ) return UNBOUNDED_SET;

    function DIFFERENCE( S1, S2 : UNBOUNDED_SET ) return UNBOUNDED_SET;

    function SUBSET( S1, S2 : UNBOUNDED_SET ) return BOOLEAN;

    function EQUAL( S1, S2 : UNBOUNDED_SET ) return BOOLEAN;
```

```
    function SIZE( S : UNBOUNDED_SET ) return NATURAL;

    function INITIALIZE( A : SET_ARRAY ) return UNBOUNDED_SET;

    -- Input/output routines
    -- These routines must be instantiated by using differnt parameter
    -- procedures for each different Element_Type.
    generic
    with procedure G_PUT( X : in ELEMENT_TYPE ) is <>;
    procedure GEN_PUT( S : in UNBOUNDED_SET );

    generic
    with procedure G_GET( X : out ELEMENT_TYPE ) is <>;
    procedure GEN_GET( S : out UNBOUNDED_SET );

    generic
    with procedure G_PUT_FILE( FILE : in FILE_TYPE;
    X : in ELEMENT_TYPE ) is <>;
    procedure GEN_FILE_PUT( FILE : in FILE_TYPE; S : in UNBOUNDED_SET );

    generic
    with procedure G_GET_FILE( FILE : in FILE_TYPE;
    X : out ELEMENT_TYPE ) is <>;
    procedure GEN_FILE_GET( FILE : in FILE_TYPE; S : out UNBOUNDED_SET );


    private
    type ELEMENT;
    type UNBOUNDED_SET is access ELEMENT;
    type ELEMENT is
    record
      NODE : ELEMENT_TYPE; -- contains the element
      NEXT : UNBOUNDED_SET := null; -- used in link list
    end record;

end UNBOUNDED_SET_PKG;
```

B. BOUNDED_MULTISET

```
--Title : Generic specification for bounded_multisets
--Author : William D. Reese
--Modification: Modified by J.M. Sealander Apr 92. The original data structure kept mulitple copies
--of any repeating elements. The data structure was changed to keep a count of each element rather than
--keeping multiple copies. Get and put operations were removed from the test package and added
--to specification. Last_Element and The_Element were removed from the package specifications
--and are now local to the body. They reflect the implementation method, not the data type.
--Overflow exception added to procedure Add. Function empty_set removed
-- and replaced with empty_set constant.
--Date : October 12, 1991
--Course : CS-4530 (Prof. Luqi)
--System : UNIX
--Compiler : VERDIX

--                  ******************************
--                  *                            *
--                  *   BOUNDED_MULTISETS        *        SPEC
--                  *                            *
--                  ******************************
generic
    type ELEMENT_TYPE is private;
    MAX_SIZE : POSITIVE;

    with procedure PUT(X: ELEMENT_TYPE);
    with procedure GET(X: out ELEMENT_TYPE);

package BOUNDED_MULTISETS is
 --Purpose
 --This package provides facilities for implementing bounded multisets
 --as abstract data types (ADT). Operations provided include an empty
 --set constant, functions for adding, removing, and counting elements,
 --as well as comparison for equality between two bounded multisets.

    type BOUNDED_MULTISET is private;

--                          ..............................
--                          .                            .
--                          .      Count                 .        SPEC
--                          .                            .
--                          ..............................

    function Count (THE_BOUNDED_MULTISET : in BOUNDED_MULTISET;
                    ELEMENT_OF_INTEREST : in ELEMENT_TYPE) return NATURAL;

--                          ..............................
--                          .                            .
--                          .      Add                   .        SPEC
--                          .                            .
--                          ..............................

    procedure Add (ELEMENT_TO_BE_ADDED : in ELEMENT_TYPE;
                   THE_MULTISET : in out BOUNDED_MULTISET);

--                          ..............................
--                          .                            .
--                          .      Remove                .        SPEC
--                          .                            .
--                          ..............................
```

```
    procedure Remove (ELEMENT_TO_BE_REMOVED : in ELEMENT_TYPE;
                      THE_MULTISET : in out BOUNDED_MULTISET);

--                    .............................
--                    .                           .
--                    .      Equals               .        SPEC
--                    .                           .
--                    .............................

    function Equals (SET_1, SET_2 : in BOUNDED_MULTISET) return BOOLEAN;

--                    .............................
--                    .                           .
--                    .      Size                 .        SPEC
--                    .                           .
--                    .............................

    function Size (THE_MULTISET : in BOUNDED_MULTISET) return NATURAL;

--                    .............................
--                    .                           .
--                    .   Put_Multiset            .        SPEC
--                    .                           .
--                    .............................

    procedure Put_Multiset (MULTISET : in BOUNDED_MULTISET);

--                    .............................
--                    .                           .
--                    .   Get_Multiset            .        SPEC
--                    .                           .
--                    .............................

    function Get_Multiset (NUMBER_OF_ELEMENTS : in NATURAL)
                                  return BOUNDED_MULTISET;

    NOT_FOUND : exception;
    OVERFLOW : exception;

    private
    type ELEMENT_RECORD is
     record
       ELEMENT : ELEMENT_TYPE;
       COUNT : NATURAL := 0;
     end record;
    type LIST is array (INTEGER range <>) of ELEMENT_RECORD;
    type BOUNDED_MULTISET is
     record
       THE_ELEMENTS : LIST(1..MAX_SIZE);
       HOW_FULL : NATURAL := 0;
     end record;
    EMPTY_SET : constant NATURAL := 0;
    end BOUNDED_MULTISETS;
```

C. BOUNDED_GRAPH

```
with TEXT_IO;
use TEXT_IO;

--                  ******************************
--                  *                            *
--                  *      Bounded_Graph         *      SPEC
--                  *                            *
--                  ******************************

generic

    type NODE_TYPE is private;
    max_size : integer;

 package Bounded_Graph is

-- Purpose
-- The routines in this package deal with a directed graphs with a number of
-- nodes <= max_size, which is established at instantiation. The package
-- specification contaions the followings functions and procedures.
--
-- add_a_node
-- add_an_edge
-- remove_an_edge
-- remove_a_node
-- nodes_connected_in
-- nodes_connected_out
-- nodes_connected
-- graph_empty
--
-- Notes
-- max_size should be only a positive integer number
-- NODE_TYPE is a user defined type
--
-- Exceptions
-- DUPLICATE_NODE -- Raised if a duplicate node is added to the graph.
-- NODE_NOT_FOUND -- Raised if a node passed in is not in the graph.
-- GRAPH_IS_FULL -- Raised if a node is added to a full graph.
-- GRAPH_IS_EMPTY -- Raised if any operation except add_a_node and graph_empty
-- is attempted on an empty graph.
--
-- Modifications
-- 11/13/91 Michael D. O'Loughlin Initial version of specification.
-- UNIX version Verdix Ada, Naval Postgraduate School
-- 02/92 J.M. Sealander Original version was a machine with state variable graph declared in
--specification. State variable removed and parameter of type Graph_Typ_Ptr added to all functions
--and procedures. An operator which returns an empty graph was added and overflow exception
--was added to a local procedure which is called by the add operation.

    type Graph_Typ_Ptr is private;

    DUPLICATE_NODE,
    NODE_NOT_FOUND,
    GRAPH_IS_FULL,
    GRAPH_IS_EMPTY : exception;
```

```
--                              .....................................
--                              .                                   .
--                              .  Graph_Empty                      .        SPEC
--                              .                                   .
--                              .....................................

    function Graph_Empty(GRAPH : in Graph_Typ_Ptr) return boolean;

-- Purpose
-- This function will check if the graph matirx is empty (no nodes on the graph)

--                              .....................................
--                              .                                   .
--                              .  Nodes_Connected                  .  SPEC
--                              .                                   .
--                              .....................................

    procedure Nodes_Connected(graph : in out Graph_Typ_Ptr;
                              a_node, b_node : in NODE_TYPE;
                              conn : out CONNECTION);

-- Purpose
-- This function will check if two two nodes are connected.

--                              .....................................
--                              .                                   .
--                              .  Add_A_Node                       .        SPEC
--                              .                                   .
--                              .....................................

    procedure Add_A_Node(graph : in out Graph_Typ_Ptr; node : in NODE_TYPE);

-- Purpose
-- This procedure will add a node to the graph (adjacency matrix).

--                              .....................................
--                              .                                   .
--                              .  Add_An_Edge                      .        SPEC
--                              .                                   .
--                              .....................................

    procedure Add_An_Edge(Graph : in out Graph_Typ_Ptr;
                          a_node, b_node : in NODE_TYPE);

-- Purpose
-- This procedure will add an edge between the two node passed in.

--                              .....................................
--                              .                                   .
--                              . Nodes_Connected_Out               .        SPEC
--                              .                                   .
--                              .....................................

    procedure Nodes_Connected_Out(graph : in out Graph_Typ_Ptr;
                                  node : in NODE_TYPE;
                                  out_nodes : out NODE_COUNT;
                                  node_connected : out CONNECTED_NODES);
```

```
-- Purpose
-- This procedure will pass out all nodes and the number of nodes connected,
-- to the node passed in, by an out edge.

--                          .....................................
--                          .                                   .
--                          .  Nodes_Connected_In  .        SPEC
--                          .                                   .
--                          .....................................

    procedure Nodes_Connected_In(Graph : in out Graph_Typ_Ptr;
                                 node : in NODE_TYPE;
                                 in_nodes : out NODE_COUNT;
                                 node_connected : out CONNECTED_NODES);

-- Purpose
-- This procedure will pass out all nodes and the number of nodes connected,
-- to the node passed in, by an in edge.

--                          .....................................
--                          .                                   .
--                          .  Remove_Edge  .        SPEC
--                          .                                   .
--                          .....................................

    procedure Remove_Edge(graph : in out Graph_Typ_Ptr;
                          a_node, b_node : in NODE_TYPE);

-- Purpose
-- This procedure will remove an edge between the two nodes passed in.

--                          .....................................
--                          .                                   .
--                          .  Remove_Node  .        SPEC
--                          .                                   .
--                          .....................................

    procedure Remove_Node(graph : in out Graph_Typ_Ptr;
                          node : in NODE_TYPE);

-- Purpose
-- This procedure will remove the node passed in from the graph and remove all its
-- related edges.

    private

    subtype NODE_COUNT is integer range 0..max_size;
    type CONNECTED_NODES is array (NODE_COUNT) of NODE_TYPE;
    type CONNECTION is (in_edge, out_edge, in_out_edge, not_connected);

    type NODE_STATUS_TYP is
     record
       empty : boolean := true; -- Empty node if true.
       node_id : NODE_TYPE; -- User defined data structure.
     end record;

    type NODE_INDEX_ARRAY_TYP is -- One dimensional array containing all nodes.
    array (NODE_COUNT) of NODE_STATUS_TYP;
    type GRAPH_MATRIX_TYP is -- Graph adjacency matrix.
    array (NODE_COUNT, NODE_COUNT) of boolean;
```

```
    type Graph_Typ is
     record
       Number_of_nodes : NODE_COUNT := 0;
       Nodes : NODE_INDEX_ARRAY_TYP;
       Graph_matrix : GRAPH_MATRIX_TYP:= (others =>(others =>true));
     end record;

    type Graph_Typ_Ptr is access GRAPH_TYP;

    empty_graph : NODE_COUNT := 0; -- Node counter for graph.


end Bounded_Graph;
```

D. UNBOUNDED_GRAPH

```
--                          *************************
--                          *                       *
--                          *   Unbounded_Graph     *     SPEC
--                          *                       *
--                          *************************

generic
    type ELEMENT_TYP is
    private;

package Unbounded_Graph is
-- PURPOSE: This generic package implements an unbounded directed
-- graph type.
-- INITIALIZATION EXCEPTIONS: none
-- NOTES: The graph is represented by a set of distinct nodes,
-- with each node having an edge list composed of a set of
-- distinct nodes. A directed edge exists between a node in the
-- graph and all nodes in it's respective edge list.
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build.
-- 05/92 modified by J.M. Sealander. An adjacency node in the original structure pointed to a graph
--node to identify an adjacent vertex. This pointer was removed and replaced with the node element. Type
--declarations used to build the data type were moved to the private section.

    type GRAPH_TYP is
    private;

    INPUT_NODE_DOES_NOT_EXIST : exception;

--                          .........................
--                          .                       .
--                          .  Is_Empty             .          SPEC
--                          .                       .
--                          .........................

    function Is_Empty(Graph : GRAPH_TYP) return BOOLEAN;

-- PURPOSE:
-- Returns TRUE if the input Graph is empty, FALSE if it is not.
-- EXCEPTIONS: none
-- NOTES: none
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build
--                          .........................
--                          .                       .
--                          .   Add_Node            .          SPEC
--                          .                       .
--                          .........................

    procedure Add_Node(New_Node : in ELEMENT_TYP; Graph : in out GRAPH_TYP);

-- PURPOSE:
-- Adds a New_Node of type ELEMENT_TYP to the input Graph.
-- EXCEPTIONS: none
-- NOTES:
-- If a node is found in the Graph which is a duplicate of
```

```
-- the New_Node, the Graph will remain unaltered.
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build

--                          .....................
--                          .                   .
--                          .  Remove_node      .       SPEC
--                          .                   .
--                          .....................

   procedure Remove_node(Node : in ELEMENT_TYP; Graph : in out GRAPH_TYP);

-- PURPOSE: Removes Node from Graph.
-- EXCEPTIONS:
-- INPUT_NODE_DOES_NOT_EXIST is raised if Node is not found in
-- Graph.
-- NOTES: All references to Node are removed from Graph, even
-- references in other Node's edge lists.
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build

--                          .....................
--                          .                   .
--                          .     Add_Edge      .       SPEC
--                          .                   .
--                          .....................

   procedure Add_Edge(From_Node : in ELEMENT_TYP;
                      To_Node : in ELEMENT_TYP;
                      Graph : in out GRAPH_TYP);

-- PURPOSE: Adds a directed edge between From_Node and To_Node
-- within Graph.
-- EXCEPTIONS:
-- INPUT_NODE_DOES_NOT_EXIST is raised if From_Node or To_Node
-- is not found in Graph.
-- NOTES: If To_Node is already an element of From_Node's edge
-- list, the graph remains unaltered.
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build

--                          .....................
--                          .                   .
--                          .  Remove_Edge      .       SPEC
--                          .                   .
--                          .....................


   procedure Remove_edge (From_Node : in ELEMENT_TYP;
                          To_Node : in ELEMENT_TYP;
                          Graph : in out GRAPH_TYP);

-- PURPOSE: Removes an edge between From_Node and To_Node in the
-- given Graph.
-- EXCEPTIONS:
-- INPUT_NODE_DOES_NOT_EXIST is raised if From_Node is not
-- found in Graph or if To_Node is not found in From_Node's edge
-- list.
-- NOTES: none
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build
```

```
--                          .............................
--                          .                           .
--                          . Has_An_Edge               .          SPEC
--                          .                           .
--                          .............................

   function Has_An_Edge (From_Node : ELEMENT_TYP;
                         To_Node : ELEMENT_TYP;
                         Graph : GRAPH_TYP)
   return BOOLEAN;

-- PURPOSE: Determines if an edge exists between the two input
-- nodes of the given Graph.
-- EXCEPTIONS:
-- INPUT_NODE_DOES_NOT_EXIST is raised if From_Node or To_Node
-- is not found in Graph.
-- NOTES: none
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build

--                          .................................................
--                          .                                               .
--                          . Set_Of_Outgoing_Edges                         .          SPEC
--                          .                                               .
--                          .................................................

   function Set_Of_Outgoing_Edges (From_Node : ELEMENT_TYP;
                                   Graph : GRAPH_TYP) return NODE_SET_PTR;

-- PURPOSE: Returns the set of edges outgoing from the given
-- From_Node in Graph.
-- EXCEPTIONS:
-- INPUT_NODE_DOES_NOT_EXIST is raised if From_Node
-- is not found in Graph.
-- NOTES: An outgoing edge set is composed of the set of all
-- nodes which have directed edges coming from the From_Node to
-- those in the set.
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build

--                          .................................................
--                          .                                               .
--                          . Set_Of_Incoming_Edges                         .          SPEC
--                          .                                               .
--                          .................................................

   function Set_Of_Incoming_Edges (To_Node : ELEMENT_TYP;
                                   Graph : GRAPH_TYP) return NODE_SET_PTR;

-- PURPOSE: Returns the set of edges incoming to the given To_Node
-- in Graph.
-- EXCEPTIONS:
-- INPUT_NODE_DOES_NOT_EXIST is raised if To_Node
-- is not found in Graph.
-- NOTES: An incoming edge set is composed of the set of all
-- nodes which have directed edges from themselves to the
-- To_Node.
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build
```

```
--                          .....................................
--                          .                                   .
--                          .   Set_Of_All_Nodes    .       SPEC
--                          .                                   .
--                          .....................................

    function Set_Of_All_Nodes (Graph : GRAPH_TYP)  return NODE_SET_PTR;

-- PURPOSE: Returns the set of all nodes in the input Graph.
-- EXCEPTIONS: none
-- NOTES: The returned node set is simply the complete set of
-- nodes in Graph, and contains no interrelationship data.
-- MODIFICATIONS:
-- 11/15/91 J.L. Budnick Initial build

    private
    type GRAPH_NODE;
    type GRAPH_TYP is
    access GRAPH_NODE;

    type EDGE;
    type EDGE_PTR is
    access EDGE;

    type EDGE is
     record
       Edge_node : ELEMENT_TYP;
       Next_Edge : EDGE_PTR := null;
     end record;

    type GRAPH_NODE is
     record
       Element : ELEMENT_TYP;
       Next_Node : GRAPH_TYP := null;
       Edges : EDGE_PTR := null;
     end record;

    EMPTY_GRAPH : constant GRAPH_TYP := null;

end Unbounded_Graph;
```

E. UNBOUNDED_MAP

```
-- Name: Capt Robert M. Dixon
-- Date: 15 Nov 91
-- Modifications : Modified by J. M. Sealander Feb 92. In the orignial version
-- NUM_BINDINGS and NUM_BLOCKS were state variables declared
-- in the package body. Erroneous data resulted if more than
-- one variable of MAP_TYPE was declared. They are now part
-- of the MAP_TYPE data structure.
-- Course: CS4530
-- Compiler: Verdix Ada 6.0
-- System: Solbourne
-- Title: Map_Package
-- Remarks: Map_Package implements an unbounded map using a generic Ada package.
-- The package is instantiated with a hash function, which returns an index
-- into the hash table, based on the domain value. If the index lies outside
-- the range of the map, it links in new map blocks until the index falls
-- on a block in the map.
--
-- The domain/range value is added to the head of a linked list whose head
-- pointer is stored in the map block, at the index pointed to by the hash
-- function.

-- Generic Parameters: DOMAIN_TYPE is the desired type of the domain.
-- RANGE_TYPE is the desired type of the range.
-- Hash is the user's hashing function.
-- Domain_Put is a put procedure for the domain type.
-- Range_Put is a put procedure for the range type.
-- Domain_Get is a get procedure for the domain type.
-- Range_Get is a get procedure for the range type.
-- Granularity specifies how many values will be stored in
-- each block of the hash table.

generic
    type DOMAIN_TYPE is private;
    type RANGE_TYPE is private;
    with function Hash(Domain_Value : in DOMAIN_TYPE) return NATURAL;
    with procedure Domain_Put(Domain_Value : in DOMAIN_TYPE);
    with procedure Range_Put(Range_Value : in RANGE_TYPE);
    with procedure Domain_Get(Domain_Value : out DOMAIN_TYPE);
    with procedure Range_Get(Range_Value : out RANGE_TYPE);
    Granularity : in INTEGER := 100;
package Map_Package is

-- MAP_TYPE is the type that the package user will use to declare a map.
-- MAP_BINDINGS_TYPE is the type used to store the number of bindings in the map.
-- Add adds a new domain/range pair to the map.
-- Range_Value returns the range value associated with the domain_value parameter.
-- Is_Bound returns true if the domain_value is in the table.
-- Number_Of_Bindings returns the number of bindings in the map.
-- Remove_Binding removes a domain/range pair from the map.
-- Put prints out the entire map.
-- Get gets the entire map.

-- Empty_Map is an empty map constant.

-- Domain_Exists is raised if a duplicate domain is inserted into the map.
-- Domain_Not_Found is raised if a domain is not found in the map.
```

```
    type MAP_TYPE is private;
    type MAP_BINDINGS_TYPE is new NATURAL;

    procedure Add(Map : in out MAP_TYPE; Domain_Value : in DOMAIN_TYPE;
                  Range_Value : in RANGE_TYPE);
    function Range_Value(Map : MAP_TYPE;  Domain_Value : DOMAIN_TYPE)
                                    return RANGE_TYPE;
    function Is_Bound(Map : MAP_TYPE; Domain_Value : DOMAIN_TYPE) return BOOLEAN;
    function Number_Of_Bindings(Map : MAP_TYPE) return MAP_BINDINGS_TYPE;
    procedure Remove_Binding(Map : in MAP_TYPE;  Domain_Value : DOMAIN_TYPE);
    procedure Put(Map : in MAP_TYPE);
    procedure Get(Map : in out MAP_TYPE);

    Empty_Map : constant MAP_TYPE;

    Domain_Exists : exception;
    Domain_Not_Found : exception;

private
    type MAP_RECORD_TYPE is
     record
       Domain_Value : DOMAIN_TYPE;
       Range_Value : RANGE_TYPE;
     end record;
    type MAP_RECORD_NODE_TYPE;
    type MAP_RECORD_NODE_PTR_TYPE is access MAP_RECORD_NODE_TYPE;
    type MAP_RECORD_NODE_TYPE is
     record
       Map_Record : MAP_RECORD_TYPE;
       Next_Node : MAP_RECORD_NODE_PTR_TYPE;
     end record;
    type MAP_ARRAY_TYPE is array(0 .. Granularity - 1) of MAP_RECORD_NODE_PTR_TYPE;
    type MAP_BLOCK_TYPE;
    type MAP_BLOCK_PTR_TYPE is access MAP_BLOCK_TYPE;
    type MAP_BLOCK_TYPE is
     record
       Map_Array : MAP_ARRAY_TYPE;
       Next_Block : MAP_BLOCK_PTR_TYPE;
     end record;
    type MAP_HEAD_TYPE;
    type MAP_TYPE_PTR is access MAP_HEAD_TYPE;
    type MAP_HEAD_TYPE is
     record
       NUM_BINDINGS : MAP_BINDINGS_TYPE := 0;
       NUM_BLOCKS : NATURAL :=0;
       HEAD : MAP_BLOCK_PTR_TYPE;
     end record;
    type MAP_TYPE is new MAP_TYPE_PTR;
    Empty_Map : constant MAP_TYPE := new MAP_HEAD_TYPE;
end Map_Package;
```

F. REAL_NUMBERS

```
-------------------------------------------------------------------
--Author original version : Dogan Ozdemir
--Author modified version : J.M. Sealander
--Original version used separate arrays to hold the whole and decimal part
--of a real number. Modified version uses one array to hold all the digits
--and the output is normalized so that the decimal point appears after the
--first digit.
--Local procedures eliminated from original version: ADD_WHOLE, SUBTRACT_WHOLE,
--M_SHIFT_LEFT, SHIFT_LEFT, SIMPLIFY.
with TEXT_IO;

generic

    DIGIT : INTEGER := 10;
    MAX_EXP : INTEGER := 3;

package REAL_PKG is

    type REAL is private;
    INPUT_ERROR : EXCEPTION;

 --functions--

    function ADDITION (NUM1,NUM2 : REAL) return REAL;
    function "+" (NUM1,NUM2 : REAL) return REAL renames ADDITION;
-------------------------------------------------------------------
    function SUBTRACTION (NM1,NM2 : REAL) return REAL;
    function "-" (NM1,NM2 : REAL) return REAL renames SUBTRACTION;
-------------------------------------------------------------------
    function MULTIPLICATION (NM1,NM2 : REAL) return REAL;
    function "*" (NM1,NM2 : REAL) return REAL renames MULTIPLICATION;
-------------------------------------------------------------------
    function DIVISION (NM1,NM2 : REAL) return REAL ;
    function "/" (NM1,NM2 : REAL) return REAL renames DIVISION;
-------------------------------------------------------------------
    function EQUAL(N1,N2 :REAL) return BOOLEAN;
-------------------------------------------------------------------
    function GREATER (N1,N2 :REAL) return BOOLEAN;
    function ">" (N1,N2 :REAL) return BOOLEAN renames GREATER;
-------------------------------------------------------------------
-- This procedure gets a float and converts it to the respective
-- real number.

    function CONV_REAL (FL : FLOAT) return REAL;
-------------------------------------------------------------------
 -- procedures--

-------------------------------------------------------------------
-- This procedure reads the Real Number from the screen and
-- decompose it into the sign, digits and exponent
-- arrays.

    procedure GET (NUM : in out REAL );
-------------------------------------------------------------------
-- This procedure puts the real number to the screen

    procedure PUT (R : in REAL);
-------------------------------------------------------------------
```

```
private

    type SIGN is ('+','-');
    subtype DECIMALS is INTEGER range 0 .. 9;
    type MANTISSA_ARRAY is array (1 .. DIGIT) of DECIMALS;
    type EXPONENT_ARRAY is array (1 .. MAX_EXP) of DECIMALS;

    type REAL is
     record
       SIGN_WHOLE : SIGN := '+';
       MANTISSA : MANTISSA_ARRAY := (others => 0);
       SIGN_EXP : SIGN := '+';
       EXPONENT : EXPONENT_ARRAY := (others => 0);
     end record;

    EMPTY_EXP : constant EXPONENT_ARRAY := (others => 0);
end REAL_PKG;
```

G. BOUNDED_INTEGERS

```
--*************************************************************************
--* TITLE : IMPLEMENTATION OF BOUNDED INTEGERS
--* COURSE : CS 4530
--* AUTHOR : Metin Balci
--* DATE : OCT,25,91
--* MODIFICATIONS : Modified by J. M. Sealander Apr 92. Subprogram algorithms simplified.
--* SYSTEM : UNIX
--* COMPILER : VERDIXADA
--* FILE : balci/qtr5/cs4530/big_int_spec.a
--* DESCRIPTION : This generic package contains the specifications
--* for implementing the type bounded integer.The
--* type is implemented as generic big integers with
--* array representations.The generic parameter is
--* DIGIT and it specifies the number of decimal digits
--* the represantation support.The generic parameter
--* is used as value generic parameter and intialized
--* to 20 as default value.You can change the generic
--* value in your implementation(or test)program.
--* By using "renames" features of ADA, the operations
--* are overloaded for the normal operators.
--* For equality check although a function "equals" is
--* supported, user can use "=" operator for this aim.
--* This feature is implemented in the test program.
--*************************************************************************

with TEXT_IO;
use TEXT_IO;

generic
    DIGIT : in INTEGER := 20;
package BOUNDED_INTEGER_ARRAY_PACKAGE is


    type BOUNDED_INTEGER is private;


-- for a given integer it returns a bounded integer,by calling this
-- function we may have operations with both types
    procedure CONVERT (NUM_VAL : in INTEGER ; B_NUM : out BOUNDED_INTEGER);

-- converts the string to a bounded integer type.User is supposed to
-- enter the big integer as a string
    procedure STRING_TO_BOUNDED_INTEGER (STR :in STRING ; LNG:in INTEGER;
                                         B_INT :out BOUNDED_INTEGER);

-- the addition of two bounded integers
    function ADDITION ( B_INT1 :in BOUNDED_INTEGER;
                        B_INT2 :in BOUNDED_INTEGER) return BOUNDED_INTEGER;

    function "+" (B_INT1 :in BOUNDED_INTEGER; B_INT2 :in BOUNDED_INTEGER)
                    return BOUNDED_INTEGER renames ADDITION;

-- subtraction
    function SUBTRACTION (LEFT :in BOUNDED_INTEGER; RIGHT:in BOUNDED_INTEGER)
                          return BOUNDED_INTEGER;

    function "-" (LEFT :in BOUNDED_INTEGER; RIGHT:in BOUNDED_INTEGER)
                  return BOUNDED_INTEGER renames SUBTRACTION;
```

```
-- multiplication
   function MULTIPLICATION (L:in BOUNDED_INTEGER; R:in BOUNDED_INTEGER)
                                    return BOUNDED_INTEGER ;

   function "*" (L:in BOUNDED_INTEGER; R:in BOUNDED_INTEGER)
                  return BOUNDED_INTEGER renames MULTIPLICATION ;

-- division
   function DIVISION (DIVIDENT :in BOUNDED_INTEGER;
                      DIVISOR :in BOUNDED_INTEGER) return BOUNDED_INTEGER;
   function "/" (DIVIDENT :in BOUNDED_INTEGER; DIVISOR :in BOUNDED_INTEGER)
                      return BOUNDED_INTEGER renames DIVISION ;


-- modulo operation
   function MODULO (FIRST :in BOUNDED_INTEGER; SECOND :in BOUNDED_INTEGER)
                        return BOUNDED_INTEGER;


   function "mod" (FIRST :in BOUNDED_INTEGER; SECOND :in BOUNDED_INTEGER) return
                    BOUNDED_INTEGER renames MODULO;

-- returns if two bounded integer is equal or not
   function EQUALS (LEFT,RIGHT : in BOUNDED_INTEGER )return BOOLEAN;



-- returns if the first entry greater than the second entry
   function GREATER_THAN ( X,Y : in BOUNDED_INTEGER ) return BOOLEAN;


   function ">" ( X,Y : in BOUNDED_INTEGER) return BOOLEAN renames GREATER_THAN;


-- this is the get function which is implemented for test purposes
   procedure GET ( STR_TO_BOUNDED : out BOUNDED_INTEGER);


-- this is the put function which is implemented for test purposes
   procedure PUT (BOUNDED_TO_STR : in BOUNDED_INTEGER);


private
   subtype NUM_OF_CHAR is INTEGER range 0..9;
   type SIGN_DIGIT is ('+','-');
   type B_INT_ARRAY_TYPE is array (NATURAL RANGE 0 .. DIGIT) of NUM_OF_CHAR;
   type BOUNDED_INTEGER is
    record
     SIGN: SIGN_DIGIT ;
     B_INT_ARRAY : B_INT_ARRAY_TYPE;
    end record;

end BOUNDED_INTEGER_ARRAY_PACKAGE;
```

H. VECTORS

```
--Title : Vector ADT
--Author : Jennie M. Sealander
--Date : 14 November 1991
--Course : CS-4530
--Compiler : Verdix Ada
--Description: Generic package for Abstract Data Type Vector.
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

--                          ************************
--                          *                      *
--                          *      VECTORS         *      SPEC
--                          *                      *
--                          ************************

with TEXT_IO; use TEXT_IO;
generic

    type ELEMENT_TYPE is private; --vector component type, must be a
 --numeric type
    DIMENSION : in POSITIVE; --dimension of vector type
    with function "+" (X,Y: ELEMENT_TYPE) return ELEMENT_TYPE;
 --Purpose
 --This function defines addition for the numeric ELEMENT_TYPE

    with function "-" (X,Y: ELEMENT_TYPE) return ELEMENT_TYPE;
 --Purpose
 --This function defines subtraction for the numeric ELEMENT_TYPE

    with function "*" (X,Y: ELEMENT_TYPE) return ELEMENT_TYPE;
 --Purpose
 --This function defines multiplication for the numeric ELEMENT_TYPE

    with function ZERO return ELEMENT_TYPE;
 --Purpose
 --This function defines zero for the numeric ELEMENT_TYPE

    with function SQRT(X: ELEMENT_TYPE) return ELEMENT_TYPE;
 --Purpose
 --This function defines the square root for ELEMENT_TYPE

    with procedure PUT(X: ELEMENT_TYPE);
 --Purpose
 --This procedure defines PUT for the numeric ELEMENT_TYPE

    with procedure GET(X: out ELEMENT_TYPE);
 --Purpose
 --This procedure defines GET for the numeric ELEMENT_TYPE

package VECTORS is

 type VECTOR is array(1..DIMENSION) of ELEMENT_TYPE;

 function "+" (V1,V2 : in VECTOR) return VECTOR;
 --Purpose
 --Vector addition
```

```
    function "-" (V1,V2 : in VECTOR) return VECTOR;
--Purpose
--Vector Subtraction

    function "*" (V1 : in VECTOR; S: in ELEMENT_TYPE) return VECTOR;
--Purpose
--Multiplication of a vector by a scalar

    function "*" (V1,V2 : in VECTOR) return ELEMENT_TYPE;
--Purpose
--Vector Dot Product, multiplication of two vectors

    function LENGTH (V : in VECTOR) return ELEMENT_TYPE;
--Purpose
--Returns magnitude of vector

    procedure PUT_VECTOR (V: VECTOR);
--Purpose
--Outputs an object of type VECTOR

    procedure GET_VECTOR (V: out VECTOR);
--Purpose
--Gets an object of type VECTOR


    INPUT_ERROR : exception;
--Purpose
--Raised if error in input of type VECTOR

end VECTORS;
```

I. MATRIX

```
--<>--<>--<>
```

APPENDIX B. KODIAK PROGRAM LISTING

```
!definition of lexical classes

%define Digit          :[0-9]
%define Int            :{Digit}+
%define Lower          :[a-z]
%define Upper          :[A-Z]
%define Letter         :({Lower}|{Upper})
%define Alpha          :({Letter}|{Digit})
%define Underscore     :["_"]
%define Blank          :[ \t\n]
%define Quote          :["]
%define Backslash      :"\\"
%define Char           :([^"\\]|{Backslash}{Quote}|{Backslash}{Backslash})

!definition of white space comments

                       :{Blank}+
                       :"--".*"\n"

!definitions of compound symbols and keywords

PACKAGE                :package|PACKAGE
IS                     :is|IS
PRIVATE                :private|PRIVATE
END                    :end|END
USE                    :use|USE
TYPE                   :type|TYPE
PROCEDURE              :procedure|PROCEDURE
FUNCTION               :function|FUNCTION
RETURN                 :return|RETURN
IN                     :in|IN
OUT                    :out|OUT
TASK                   :task|TASK
ENTRY                  :entry|ENTRY
EXCEPTION              :exception|EXCEPTION
RENAMES                :renames|RENAMES
CONSTANT               :constant|CONSTANT|""
SUBTYPE                :subtype|SUBTYPE
NEW                    :new|NEW
RANGE                  :range|RANGE
GENERIC                :generic|GENERIC
WITH                   :with|WITH
DIGITS                 :digits|DIGITS
DELTA                  :delta|DELTA
LIMITED                :limited|LIMITED
FOR                    :for|FOR
AT                     :at|AT
ALL                    :all|ALL
CASE                   :case|CASE
WHEN                   :when|WHEN
OTHERS                 :others|OTHERS
ACCESS                 :access|ACCESS
AND                    :"&"|"and"|"AND"
THEN                   :then|THEN
OR                     :"|"|"or"|"OR"
ELSE                   :else|ELSE
NOT                    :not|NOT
```

```
XOR                     :xor|XOR
ABS                     :abs|ABS
NULL                    :null|NULL
EQUAL                   :"="|":="|""
NEQ                     :"/="
LT                      :"<"
LTE                     :"<="
GTE                     :">="
GT                      :">"
PLUS                    :"+"
MINUS                   :"-"
TIMES                   :"*"
DIVIDE                  :"/"
EXPONENT                :"**"
MOD                     :mod|MOD
REM                     :rem|REM
ARRAY                   :array|ARRAY
OF                      :of|OF
RECORD                  :record|RECORD
DISCRETE                :"<>"
ARROW                   :"=>"
TO                      :".."
TIC                     :"'"
CHARACTER_LITERAL       :"'".""
STRING_LITERAL          :{Quote}{Char}*{Quote}
INTEGER_LITERAL         :{Int}
REAL_LITERAL            :{Int}"."{Int}
IDENTIFIER              :{Letter}+(({Underscore}|Alpha)*{Alpha})*


%%
!Explanations of attributes

!psdl_interface_specification : synthesized string, the result of the
!                       translation

!operator_specification : synthesized string, builds the operator specification
!                   for psdl types

!number_of_operators : synthesized integer, counts the number of operators
!               in ada specification to determine if psdl interface
!               is a type or single operator

!new_composite_types : synthesized map, used to build the inherited map,
!                composite_types

!composite_types : inherited map, used to determine if a generic declaration
!               is part of a composite type declaration, i.e. an array type

!new_generic_types : synthesized map, used to build the inherited map,
!               generic_types

!generic_types : inherited map, provides the type names of the
!              index and element types for a generic  array type

!generic_type_declarations : synthesized string, builds the generic type
!                   declaration portion of psdl then inherited
!                   by package_specification

!type_declarations : synthesized string, builds the non-generic type
!               declaration portion of psdl
```

```
!input_parameters : synthesized string, builds the input attribute of a psdl
!                   operator

!output_parameters : synthesized string, builds the output attribute of a psdl
!                   operator

!mode: synthesized map, used to determine if there are any input out output
!     parameters to an operator specification

!mode_check: inherited map, initializes attribute mode to default of empty
!           string

!current_mode: synthesized string, used to determine if a comma is required
!             between two parameters

!exceptions : synthesized string, provides exceptions declared in a single
!           operator package

!variable_name : synthesized string, provides the name of an input or output
!               parameter

!variable_type : synthesized string, provides the type name for variables
!               declared in the generic portion and type declaration of the
!               PSDL specification

!attribute declarations for nonterminal symbols

start{ psdl_interface_specification:string; };

ada_interface{ psdl_interface_specification:string;
           file_name:string; };

generic_specification{ psdl_interface_specification:string;
                  generic_type_declarations:string;
                  file_name:string; };

generic_formal_part{ generic_type_declarations:string; };

generic_parameter_declarations{generic_type_declarations:string;
                       new_composite_types:string->string;
                       composite_types:string->string;
                       new_generic_types:string->string;
                       generic_types:string->string;
                       comma:string; };

generic_parameter_declaration{generic_type_declarations:string;
                      new_composite_types:string->string;
                      composite_types:string->string;
                      new_generic_types:string->string;
                      generic_types:string->string;
                      comma:string; };

generic_type_definition{variable_type:string;
                  generic_types:string->string;
                  new_composite_types:string->string; };

private_type_declaration{generic_type_declarations:string;
                   type_declarations:string; };

package_specification{ psdl_interface_specification:string;
```

```
                number_of_operators:int;
                generic_type_declarations:string;
                file_name:string; };

basic_declarative_items{ type_declarations:string;
                operator_specification:string;
                number_of_operators:int;
                input_parameters:string;
                output_parameters:string;
                exceptions:string; };

basic_declarative_item{ type_declarations:string;
                operator_specification:string;
                input_parameters:string;
                output_parameters:string;
                exceptions:string;
                number_of_operators:int; };

basic_declaration{ type_declarations:string;
            operator_specification:string;
            input_parameters:string;
            output_parameters:string;
            exceptions:string;
            number_of_operators:int; };

subprogram_declaration{ operator_specification:string;
                input_parameters:string;
                output_parameters:string;
                number_of_operators:int;
                exceptions:string; };

subprogram_specification{ operator_specification:string;
                input_parameters:string;
                output_parameters:string;
                number_of_operators:int;
                name:string; };

formal_part{ input_parameters:string;
        output_parameters:string; };

designator{ name:string;
        operator_symbols:string->string; };

operator_symbol{ name:string; };

parameter_specifications{ input_parameters:string;
                output_parameters:string;
                mode:string->string;
                mode_check:string->string;
                current_mode:string; };

parameter_specification{ input_parameters:string;
                output_parameters:string;
                mode:string->string;
                mode_check:string->string;
                current_mode:string; };

type_declarations{ type_declarations:string; };

type_declaration{ type_declarations:string; };
```

```
full_type_declaration{ type_declarations:string; };

exception_declaration{ exceptions:string; };

subtype_declaration{ type_declarations:string; };

generic_declaration{ type_declarations:string; };

subtype_indication{ variable_type:string;
            generic_types:string->string;
            new_composite_types:string->string;
            element_type:string; };

type_definition{ variable_type:string; };

real_type_definition{ variable_type:string; };

array_type_definition{ variable_type:string;
              generic_types:string->string;
              new_composite_types:string->string; };

unconstrained_array_definition{ variable_type:string;
                    generic_types:string->string;
                    new_composite_types:string->string; };

constrained_array_definition{ variable_type:string;
                  generic_types:string->string;
                  new_composite_types:string->string; };

subtype_definitions{ generic_types:string->string;
            new_composite_types:string->string;
            index_type:string; };

index_subtype_definition{ generic_types:string->string;
                index_type:string;
                new_composite_types:string->string; };

identifier_list{ variable_names:string;
          exceptions:string; };

type_mark{ variable_type:string; };

name{variable_type:string;
   variable_name:string;
   generic_types:string->string;
   new_composite_types:string->string;
   index_type:string;
   element_type:string; };


!attribute declarations for terminal symbols

IDENTIFIER{ %text:string; };
STRING_LITERAL{ %text:string; };

%%
!Productions of the grammar

start
     : ada_interface
        { %output(ada_interface.psdl_interface_specification);
```

```
        %outfile(ada_interface.file_name,
            ada_interface.psdl_interface_specification); }
    ;

ada_interface
    : context_clause generic_specification
        { ada_interface.psdl_interface_specification =
            generic_specification.psdl_interface_specification;
          ada_interface.file_name =
            generic_specification.file_name; }
    | context_clause package_specification
        { ada_interface.psdl_interface_specification =
            package_specification.psdl_interface_specification;
          ada_interface.file_name =
            package_specification.file_name; }
    ;

context_clause
    : with_clauses use_clauses
        {}
    |
        {}
    ;

with_clauses
    : with_clauses with_clause
        {}
    | with_clause
        {}
    ;

use_clauses
    : use_clauses use_clause
        {}
    |
        {}
    ;

with_clause
    : WITH package_names ';'
        {}
    ;

use_clause
    : USE package_names ';'
        {}
    ;

package_names
    : package_names ',' name
        {}
    | name
        {}
    ;

generic_specification
    : generic_formal_part subprogram_specification
        { }
    | generic_formal_part package_specification
        {package_specification.generic_type_declarations =
            generic_formal_part.generic_type_declarations;
```

```
        generic_specification.psdl_interface_specification =
            package_specification.psdl_interface_specification;
        generic_specification.file_name =
            package_specification.file_name; }
    ;

generic_formal_part
    : GENERIC generic_parameter_declarations
        {generic_formal_part.generic_type_declarations =
          [" GENERIC\n",
            generic_parameter_declarations.generic_type_declarations,
          "\n"];
        generic_parameter_declarations.generic_types =
          generic_parameter_declarations.new_generic_types
            +| {(?:string:"INDEX_TYPE")};
        generic_parameter_declarations.composite_types =
          generic_parameter_declarations.new_composite_types
            +| {(?:string:"no")}; }
    |
        {}
    ;

generic_parameter_declarations
    : generic_parameter_declarations generic_parameter_declaration
        { generic_parameter_declarations[1].generic_type_declarations =
            generic_parameter_declarations[2].comma == "yes"
          ->[generic_parameter_declarations[2].generic_type_declarations,
            ",\n",
            generic_parameter_declaration.generic_type_declarations]
          # [generic_parameter_declarations[2].generic_type_declarations,
            generic_parameter_declaration.generic_type_declarations];
          generic_parameter_declarations[1].new_generic_types =
            [generic_parameter_declarations[2].new_generic_types +|
            generic_parameter_declaration.new_generic_types];
          generic_parameter_declaration.generic_types =
            generic_parameter_declarations.generic_types;
          generic_parameter_declarations[2].generic_types =
            generic_parameter_declarations[1].generic_types;
          generic_parameter_declarations[1].new_composite_types =
            generic_parameter_declarations[2].new_composite_types
            +| generic_parameter_declaration.new_composite_types;
          generic_parameter_declaration.composite_types =
            generic_parameter_declarations.composite_types;
          generic_parameter_declarations[2].composite_types =
            generic_parameter_declarations[1].composite_types;
          generic_parameter_declarations[1].comma =
            generic_parameter_declaration.comma; }
    | generic_parameter_declaration
        { generic_parameter_declarations.generic_type_declarations =
            generic_parameter_declaration.generic_type_declarations;
          generic_parameter_declaration.generic_types =
            generic_parameter_declarations.generic_types;
          generic_parameter_declaration.composite_types =
            generic_parameter_declarations.composite_types;
          generic_parameter_declarations.new_composite_types =
            generic_parameter_declaration.new_composite_types;
          generic_parameter_declarations.new_generic_types =
            generic_parameter_declaration.new_generic_types;
          generic_parameter_declarations.comma =
            generic_parameter_declaration.comma; }
    ;
```

```
generic_parameter_declaration
    : identifier_list ':' type_mark ';'
        {generic_parameter_declaration.generic_type_declarations =
          ["     ",identifier_list.variable_names,": GENERIC_VALUE"];
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.new_generic_types = "";
         generic_parameter_declaration.comma = "yes"; }
    |identifier_list ':' IN type_mark ';'
        {generic_parameter_declaration.generic_type_declarations =
          ["     ",identifier_list.variable_names,": GENERIC_VALUE"];
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.new_generic_types = "";
         generic_parameter_declaration.comma = "yes"; }
    | identifier_list ':' IN OUT type_mark ';'
        {generic_parameter_declaration.generic_type_declarations =
          ["     ",identifier_list.variable_names,": GENERIC_VALUE"];
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.new_generic_types = "";
         generic_parameter_declaration.comma = "yes"; }
    | identifier_list ':' type_mark EQUAL expression ';'
        {generic_parameter_declaration.generic_type_declarations =
          ["     ",identifier_list.variable_names,": GENERIC_VALUE"];
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.new_generic_types = "";
         generic_parameter_declaration.comma = "yes"; }
    | identifier_list ':' IN type_mark EQUAL expression ';'
        {generic_parameter_declaration.generic_type_declarations =
          ["     ",identifier_list.variable_names,": GENERIC_VALUE"];
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.new_generic_types = "";
         generic_parameter_declaration.comma = "yes"; }
    | identifier_list ':' IN OUT type_mark EQUAL expression ';'
        {generic_parameter_declaration.generic_type_declarations =
          ["     ",identifier_list.variable_names,": GENERIC_VALUE"];
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.new_generic_types = "";
         generic_parameter_declaration.comma = "yes"; }
    | TYPE IDENTIFIER IS PRIVATE ';'
        {generic_parameter_declaration.generic_type_declarations =
          generic_parameter_declaration.composite_types(IDENTIFIER.%text)
            == "yes"
          -> ""
          # ["     ",IDENTIFIER.%text," : GENERIC_TYPE"];
         generic_parameter_declaration.new_generic_types =
          {(IDENTIFIER.%text:"PRIVATE")};
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.comma =
          generic_parameter_declaration.composite_types(IDENTIFIER.%text)
            == "yes"
          -> "no"
          # "yes"; }
    | TYPE IDENTIFIER IS discriminant_part PRIVATE ';'
        {generic_parameter_declaration.generic_type_declarations =
          generic_parameter_declaration.composite_types(IDENTIFIER.%text)          == "yes"
          -> ""
          #["     ",IDENTIFIER.%text," : GENERIC_TYPE"];
         generic_parameter_declaration.new_generic_types =
          {(IDENTIFIER.%text:"PRIVATE")};
         generic_parameter_declaration.new_composite_types = "";
         generic_parameter_declaration.comma =
```

```
      generic_parameter_declaration.composite_types(IDENTIFIER.%text)
        == "yes"
      -> "no"
      # "yes"; }
| TYPE IDENTIFIER IS LIMITED PRIVATE ';'
    {generic_parameter_declaration.generic_type_declarations =
      generic_parameter_declaration.composite_types(IDENTIFIER.%text)
        == "yes"
      -> ""
      #["     ",IDENTIFIER.%text," : GENERIC_TYPE"];
     generic_parameter_declaration.new_generic_types =
      {(IDENTIFIER.%text:"PRIVATE")};
     generic_parameter_declaration.new_composite_types = "";
     generic_parameter_declaration.comma =
      generic_parameter_declaration.composite_types(IDENTIFIER.%text)
        == "yes"
      -> "no"
      # "yes"; }
| TYPE IDENTIFIER discriminant_part IS LIMITED PRIVATE ';'
    {generic_parameter_declaration.generic_type_declarations =
      generic_parameter_declaration.composite_types(IDENTIFIER.%text)
        == "yes"
      -> ""
      #["     ",IDENTIFIER.%text," : GENERIC_TYPE"];
     generic_parameter_declaration.new_generic_types =
      {(IDENTIFIER.%text:"PRIVATE")};
     generic_parameter_declaration.new_composite_types = "";
     generic_parameter_declaration.comma =
      generic_parameter_declaration.composite_types(IDENTIFIER.%text)
        == "yes"
      -> "no"
      # "yes"; }
| TYPE IDENTIFIER IS generic_type_definition ';'
    {generic_parameter_declaration.generic_type_declarations =
      generic_parameter_declaration.composite_types(IDENTIFIER.%text)
        == "yes"
      -> ""
      #["     ",IDENTIFIER.%text," : ",
        generic_type_definition.variable_type];
     generic_type_definition.generic_types =
      generic_parameter_declaration.generic_types;
     generic_parameter_declaration.new_generic_types =
      {(IDENTIFIER.%text:"DISCRETE")};
     generic_parameter_declaration.new_composite_types =
      generic_type_definition.new_composite_types;
     generic_parameter_declaration.comma =
      generic_parameter_declaration.composite_types(IDENTIFIER.%text)
        == "yes"
      -> "no"
      # "yes";}
| WITH subprogram_specification ';'
    { generic_parameter_declaration.generic_type_declarations =
       ["     ",subprogram_specification.name,": GENERIC_PROCEDURE"];
     generic_parameter_declaration.new_composite_types = "";
     generic_parameter_declaration.new_generic_types = "";
     generic_parameter_declaration.comma = "yes"; }
| WITH subprogram_specification IS name ';'
    { generic_parameter_declaration.generic_type_declarations =
       ["     ",subprogram_specification.name,": GENERIC_PROCEDURE"];
     generic_parameter_declaration.new_composite_types = "";
     generic_parameter_declaration.new_generic_types = "";
```

```
        generic_parameter_declaration.comma = "yes"; }
    | WITH subprogram_specification IS DISCRETE ';'
        { generic_parameter_declaration.generic_type_declarations =
            ["     ",subprogram_specification.name,": GENERIC_PROCEDURE"];
          generic_parameter_declaration.new_composite_types = "";
          generic_parameter_declaration.new_generic_types = "";
          generic_parameter_declaration.comma = "yes"; }
    ;


package_specification
    : PACKAGE IDENTIFIER IS basic_declarative_items END IDENTIFIER ';'
      { package_specification.psdl_interface_specification =
          basic_declarative_items.number_of_operators == 1
        -> ["OPERATOR ",IDENTIFIER.%text,"\nSPECIFICATION\n",
            package_specification.generic_type_declarations,"\n",
            basic_declarative_items.input_parameters,
            basic_declarative_items.output_parameters,"\n",
            basic_declarative_items.exceptions,
          "END\n"]
        # ["TYPE ",IDENTIFIER.%text,"\nSPECIFICATION\n",
            package_specification.generic_type_declarations,"\n",
            basic_declarative_items.type_declarations,
            basic_declarative_items.operator_specification,
          "END\n"];
        package_specification.file_name = [IDENTIFIER.%text, ".psdl"]; }
    | PACKAGE IDENTIFIER IS basic_declarative_items PRIVATE
              basic_declarative_items END IDENTIFIER ';'
      { package_specification.psdl_interface_specification =
          basic_declarative_items.number_of_operators == 1
        -> ["OPERATOR ",IDENTIFIER.%text,"\nSPECIFICATION\n",
            package_specification.generic_type_declarations,"\n",
            basic_declarative_items.input_parameters,
            basic_declarative_items.output_parameters,
            basic_declarative_items.exceptions,
          "END\n"]
        # ["TYPE ",IDENTIFIER.%text,"\nSPECIFICATION\n",
            package_specification.generic_type_declarations,"\n",
            basic_declarative_items.type_declarations,
            basic_declarative_items.operator_specification,
          "END\n"];
        package_specification.file_name = [IDENTIFIER.%text, ".psdl"]; }
    ;

basic_declarative_items
    : basic_declarative_items basic_declarative_item
      { basic_declarative_items[1].type_declarations =
          [ basic_declarative_items[2].type_declarations,
              basic_declarative_item.type_declarations];
        basic_declarative_items[1].operator_specification =
          [ basic_declarative_items[2].operator_specification,
              "\n",basic_declarative_item.operator_specification];
        basic_declarative_items[1].input_parameters =
          [ basic_declarative_items[2].input_parameters,
              basic_declarative_item.input_parameters];
        basic_declarative_items[1].output_parameters =
          [ basic_declarative_items[2].output_parameters,
              basic_declarative_item.output_parameters];
        basic_declarative_items[1].exceptions =
          [ basic_declarative_items[2].exceptions,
              basic_declarative_item.exceptions];
```

```
        basic_declarative_items[1].number_of_operators =
            basic_declarative_items[2].number_of_operators +
            basic_declarative_item.number_of_operators; }
    |
        { basic_declarative_items.type_declarations = "";
          basic_declarative_items.operator_specification = "";
          basic_declarative_items.number_of_operators = 0;
          basic_declarative_items.input_parameters = "";
          basic_declarative_items.output_parameters = "";
          basic_declarative_items.exceptions = ""; }
    ;

basic_declarative_item
    : basic_declaration
      { basic_declarative_item.type_declarations =
        basic_declaration.type_declarations;
       basic_declarative_item.operator_specification =
        basic_declaration.operator_specification;
       basic_declarative_item.input_parameters =
        basic_declaration.input_parameters;
       basic_declarative_item.output_parameters =
        basic_declaration.output_parameters;
       basic_declarative_item.exceptions =
        basic_declaration.exceptions;
       basic_declarative_item.number_of_operators =
        basic_declaration.number_of_operators; }
    | representation_clause
      {}
    | use_clause
      {}
    ;

basic_declaration
    : object_declaration
      { basic_declaration.number_of_operators = 0;
       basic_declaration.type_declarations = "";
       basic_declaration.operator_specification = ""; }
    | type_declaration
      { basic_declaration.type_declarations =
        type_declaration.type_declarations;
       basic_declaration.operator_specification = "";
       basic_declaration.number_of_operators = 0; }
    | subprogram_declaration
      { basic_declaration.type_declarations = "";
       basic_declaration.operator_specification =
        subprogram_declaration.operator_specification;
       basic_declaration.input_parameters =
        subprogram_declaration.input_parameters;
       basic_declaration.output_parameters =
        subprogram_declaration.output_parameters;
       basic_declaration.exceptions = "";
       basic_declaration.number_of_operators =
        subprogram_declaration.number_of_operators + 1; }
    | task_declaration
      {}
    | exception_declaration
      {basic_declaration.exceptions = exception_declaration.exceptions;
       basic_declaration.type_declarations = "";
       basic_declaration.input_parameters = "";
       basic_declaration.output_parameters = "";
       basic_declaration.operator_specification = "";
```

```
        basic_declaration.number_of_operators = 0; }
    | renaming_declaration
        {basic_declaration.type_declarations = "";
        basic_declaration.operator_specification = "";
        basic_declaration.number_of_operators = 0; }
    | subtype_declaration
        {basic_declaration.type_declarations =
          subtype_declaration.type_declarations;
         basic_declaration.operator_specification = "";
         basic_declaration.number_of_operators = 0; }
    | generic_declaration
        {basic_declaration.type_declarations = "";
         basic_declaration.operator_specification = "";
         basic_declaration.number_of_operators = 0; }
    ;

representation_clause
    : type_representation_clause
        {}
    | address_clause
        {}
    ;

type_representation_clause
    : enumeration_representation_clause
        {}
    | length_clause
        {}
    | record_representation_clause
        {}
    ;

address_clause
    : FOR IDENTIFIER USE AT simple_expression ';'
        {}
    ;

enumeration_representation_clause
    : FOR IDENTIFIER USE aggregate ';'
        {}
    ;

length_clause
    : FOR attribute USE simple_expression ';'
        {}
    ;

record_representation_clause
    : FOR IDENTIFIER USE RECORD alignment_clauses component_clauses
                                    END RECORD ';'
        {}
    ;

alignment_clauses
    : alignment_clauses alignment_clause
        {}
    |
        {}
    ;

alignment_clause
```

```
    : AT MOD simple_expression ';'
      {}
    ;

component_clauses
    : component_clauses component_clause
      {}
    |
      {}
    ;

component_clause
    : name AT simple_expression RANGE range ';'
      {}
    ;

object_declaration
    : identifier_list ':' subtype_indication ';'
      {}
    | identifier_list ':' constrained_array_definition ';'
      {}
    | identifier_list ':' CONSTANT subtype_indication ';'
      {}
    | identifier_list ':' CONSTANT constrained_array_definition ';'
      {}
    | identifier_list ':' subtype_indication EQUAL expression ';'
      {}
    | identifier_list ':' CONSTANT subtype_indication EQUAL expression ';'
      {}
    | identifier_list ':' constrained_array_definition EQUAL expression ';'
      {}
    | identifier_list ':' constrained_array_definition EQUAL expression ';'
      {}
    | IDENTIFIER ':' subtype_indication ';'
      {}
    | IDENTIFIER ':' constrained_array_definition ';'
      {}
    | IDENTIFIER ':' CONSTANT subtype_indication ';'
      {}
    | IDENTIFIER ':' CONSTANT constrained_array_definition ';'
      {}
    | IDENTIFIER ':' subtype_indication EQUAL expression ';'
      {}
    | IDENTIFIER ':' CONSTANT subtype_indication EQUAL expression ';'
      {}
    | IDENTIFIER ':' constrained_array_definition EQUAL expression ';'
      {}
    | IDENTIFIER ':' constrained_array_definition EQUAL expression ';'
      {}
    ;

type_declaration
    : full_type_declaration
      { type_declaration.type_declarations = ""; }
    | incomplete_type_declaration
      { type_declaration.type_declarations = ""; }
    | private_type_declaration
      { type_declaration.type_declarations =
        private_type_declaration.type_declarations; }
    ;
```

```
subprogram_declaration
      : subprogram_specification ';'
        { subprogram_declaration.operator_specification =
          subprogram_specification.operator_specification;
         subprogram_declaration.input_parameters =
          subprogram_specification.input_parameters;
         subprogram_declaration.output_parameters =
          subprogram_specification.output_parameters;
         subprogram_declaration.number_of_operators = 0; }
      ;

task_declaration
      : task_specification ';'
        {}
      ;

exception_declaration
      : IDENTIFIER ':' EXCEPTION ';'
        {exception_declaration.exceptions =
          [" exception : ",IDENTIFIER.%text, "\n"]; }
      | identifier_list ':' EXCEPTION ';'
        {exception_declaration.exceptions =
          [" exception : ",identifier_list.exceptions, "\n"]; }
      ;

renaming_declaration
      : IDENTIFIER ':' type_mark RENAMES name ';'
        {}
      | IDENTIFIER ':' EXCEPTION RENAMES name ';'
        {}
      | PACKAGE IDENTIFIER RENAMES name ';'
        {}
      | subprogram_specification RENAMES name ';'
        {}
      ;

subtype_declaration
      : SUBTYPE IDENTIFIER IS subtype_indication ';'
        { subtype_declaration.type_declarations = ""; }
      ;

generic_declaration
      : generic_specification ';'
        {}
      ;

full_type_declaration
      : TYPE IDENTIFIER discriminant_part IS  type_definition ';'
        {}
      ;

incomplete_type_declaration
      : TYPE IDENTIFIER discriminant_part ';'
        {}
      ;

discriminant_part
      : '(' discriminant_specifications ')'
        {}
      |
        {}
```

```
    ;

discriminant_specifications
    : discriminant_specifications ';' discriminant_specification
      {}
    | discriminant_specification
      {}
    ;

discriminant_specification
    : identifier_list ':' type_mark
      {}
    | identifier_list ':' type_mark EQUAL expression
      {}
    ;
type_definition
    : enumeration_type_definition
      {}
    | real_type_definition
      {}
    | record_type_definition
      {}
    | derived_type_definition
      {}
    | integer_type_definition
      {}
    | array_type_definition
      {}
    | access_type_definition
      {}
    ;

subprogram_specification
    : PROCEDURE IDENTIFIER formal_part
      { subprogram_specification.operator_specification =
        [" OPERATOR ",IDENTIFIER.%text,"\n SPECIFICATION\n",
            formal_part.input_parameters,
            formal_part.output_parameters," END\n" ];
       subprogram_specification.input_parameters =
           [formal_part.input_parameters];
       subprogram_specification.output_parameters =
           [formal_part.output_parameters];
       subprogram_specification.name = IDENTIFIER.%text; }
    | FUNCTION designator formal_part RETURN type_mark
      { subprogram_specification.operator_specification =
        [" OPERATOR ",designator.name,"\n SPECIFICATION\n",
          formal_part.input_parameters,
          "   output ",designator.name," : ",
              type_mark.variable_type,"\n END\n" ];
       subprogram_specification.name = designator.name;
       subprogram_specification.input_parameters =
           [formal_part.input_parameters];
       subprogram_specification.output_parameters =
           ["   output ",designator.name," : ",
               type_mark.variable_type]; }
    ;

designator
    : IDENTIFIER
      { designator.name = IDENTIFIER.%text; }
    | STRING_LITERAL
```

```
        { designator.operator_symbols = {("\"+\"":"add")
          ("\"-\"":"subtract") ("\"*\"":"multiply") ("\"/\"":"divide")
          ("\"=\"":"equal") ("\"<\"":"less_than") ("\">\"":"greater_than")
          ("\"<=\"":"LTE") ("\">=\"":"GTE")
          (?:string:"overloaded_operator")};
          designator.name =
            designator.operator_symbols(STRING_LITERAL.%text); }
      ;

formal_part
      : '(' parameter_specifications ')'
        { parameter_specifications.mode_check = {(?:string:"")};
          formal_part.input_parameters =
            parameter_specifications.mode("input_parameter") == "yes"
             -> [" input ",parameter_specifications.input_parameters,
                 "\n"]
             # "";
          formal_part.output_parameters =
            parameter_specifications.mode("output_parameter") == "yes"
             -> [" output ",parameter_specifications.output_parameters,
                 "\n"]
             # ""; }
      |
        { formal_part.input_parameters = "";
          formal_part.output_parameters = ""; }
      ;

parameter_specifications
      : parameter_specifications ';' parameter_specification
        { parameter_specifications[1].input_parameters =
          ((parameter_specification.current_mode == "in") ||
            (parameter_specification.current_mode == "inout")) &&
            ((parameter_specifications[2].current_mode == "in") ||
            (parameter_specifications[2].current_mode == "inout"))
          ->[parameter_specifications[2].input_parameters,
              ", ",parameter_specification.input_parameters]
          # [parameter_specifications[2].input_parameters,
              parameter_specification.input_parameters];
         parameter_specifications[1].output_parameters =
          ((parameter_specification.current_mode == "out") ||
            (parameter_specification.current_mode == "inout")) &&
            ((parameter_specifications[2].current_mode == "out") ||
            (parameter_specifications[2].current_mode == "inout"))
          ->[parameter_specifications[2].output_parameters,
              ", ",parameter_specification.output_parameters]
          # [parameter_specifications[2].output_parameters,
              parameter_specification.output_parameters];
          parameter_specifications[1].mode =
            parameter_specifications[2].mode
             +| parameter_specification.mode;
         parameter_specifications[2].mode_check =
            parameter_specifications[1].mode_check;
         parameter_specification.mode_check =
            parameter_specifications[1].mode_check;
         parameter_specifications[1].current_mode =
            parameter_specifications[2].current_mode; }
      | parameter_specification
        { parameter_specifications.input_parameters =
          parameter_specification.input_parameters;
         parameter_specifications.output_parameters =
          parameter_specification.output_parameters;
```

```
        parameter_specifications.mode = parameter_specification.mode;
        parameter_specification.mode_check =
          parameter_specifications.mode_check;
        parameter_specifications.current_mode =
          parameter_specification.current_mode; }
    ;

parameter_specification
    : identifier_list ':' type_mark
      { parameter_specification.input_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];
        parameter_specification.output_parameters = "";
        parameter_specification.mode = {("input_parameter":"yes")}
                    +| parameter_specification.mode_check;
        parameter_specification.current_mode = "in"; }
    | identifier_list ':' IN type_mark
      { parameter_specification.input_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];
        parameter_specification.output_parameters = "";
        parameter_specification.mode = {("input_parameter":"yes")}
                    +| parameter_specification.mode_check;
        parameter_specification.current_mode = "in"; }
    | identifier_list ':' IN OUT type_mark
      { parameter_specification.input_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];
        parameter_specification.output_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];
        parameter_specification.mode = {("input_parameter":"yes"),
                          ("output_parameter":"yes")}
                    +| parameter_specification.mode_check;
        parameter_specification.current_mode = "inout"; }
    | identifier_list ':' OUT type_mark
      { parameter_specification.input_parameters = "";
        parameter_specification.output_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];
        parameter_specification.mode = {("output_parameter":"yes")}
                    +| parameter_specification.mode_check;
        parameter_specification.current_mode = "out"; }
    | identifier_list ':' type_mark EQUAL expression
      { parameter_specification.input_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];}
    | identifier_list ':' IN type_mark EQUAL expression
      { parameter_specification.input_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];}
    | identifier_list ':' IN OUT type_mark EQUAL expression
      { parameter_specification.input_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];
        parameter_specification.output_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];}
    | identifier_list ':' OUT type_mark EQUAL expression
      { parameter_specification.output_parameters =
        [identifier_list.variable_names, ": ",type_mark.variable_type];}
    ;


task_specification
    : TASK IDENTIFIER
      {}
    | TASK TYPE IDENTIFIER
      {}
    | TASK IDENTIFIER IS entry_declarations representation_clauses
```

```
                        END IDENTIFIER
            {}
    | TASK TYPE IDENTIFIER IS entry_declarations representation_clauses
                        END IDENTIFIER
            {}
    ;

entry_declarations
    : entry_declarations entry_declaration
            {}
    |
            {}
    ;

representation_clauses
    : representation_clauses representation_clause
            {}
    |
            {}
    ;

entry_declaration
    : ENTRY IDENTIFIER formal_part ';'
            {}
    | ENTRY IDENTIFIER '(' discrete_range ')' formal_part ';'
            {}
    ;

subtype_indication
    : name
            {subtype_indication.variable_type = name.variable_type;
             subtype_indication.element_type = name.element_type;
             name.generic_types = subtype_indication.generic_types;
             subtype_indication.new_composite_types =
               name.new_composite_types; }
    | name constraint
            {subtype_indication.variable_type = name.variable_type;
             subtype_indication.element_type = name.element_type;
             name.generic_types = subtype_indication.generic_types;
             subtype_indication.new_composite_types =
               name.new_composite_types; }
    ;

type_mark
    : name
            {type_mark.variable_type = name.variable_type; }
    ;

constraint
    : range_constraint
            {}
    | fixed_point_constraint
            {}
    | floating_point_constraint
            {}
    | discriminant_constraint
            {}
    | index_constraint
            {}
    ;
```

```
derived_type_definition
      : NEW subtype_indication
          {}
      ;

range_constraint
      : RANGE range
          {}
      ;

range
      : attribute
          {}
      | simple_expression TO simple_expression
          {}
      ;

discriminant_constraint
      : '(' discriminant_associations ')'
          {}
      ;

discriminant_associations
      : discriminant_associations ',' discriminant_association
          {}
      | discriminant_association
          {}
      ;

discriminant_association
      : expression
          {}
      | discriminant_choices ARROW expression
          {}
      ;

discriminant_choices
      : discriminant_choices '|' IDENTIFIER
          {}
      | IDENTIFIER
          {}
      ;

generic_type_definition
      : '(' DISCRETE ')'
          { generic_type_definition.variable_type = "DISCRETE";
            generic_type_definition.new_composite_types = ""; }
      | RANGE DISCRETE
          { generic_type_definition.variable_type = "DISCRETE";
            generic_type_definition.new_composite_types = ""; }
      | DIGITS DISCRETE
          {}
      | DELTA DISCRETE
          {}
      | array_type_definition
          { generic_type_definition.variable_type =
              array_type_definition.variable_type;
            array_type_definition.generic_types =
              generic_type_definition.generic_types;
            generic_type_definition.new_composite_types =
              array_type_definition.new_composite_types; }
```

```
    | access_type_definition
        {}
    ;

generic_instantiation
    : PACKAGE IDENTIFIER IS NEW name generic_actual_part ';'
        {}
    | PROCEDURE IDENTIFIER IS NEW name generic_actual_part ';'
        {}
    | FUNCTION designator IS NEW name generic_actual_part ';'
        {}
    ;

generic_actual_part
    : '(' generic_associations ')'
        {}
    |
        {}
    ;

generic_associations
    : generic_associations ',' generic_association
        {}
    | generic_association
        {}
    ;

generic_association
    : generic_formal_parameter ARROW generic_actual_parameter
        {}
    | generic_actual_parameter
        {}
    ;

generic_formal_parameter
    : IDENTIFIER
        {}
    | STRING_LITERAL    !operator_symbol
        {}
    ;

generic_actual_parameter
    : expression
        {}
    | name
        {}
    ;

private_type_declaration
    : TYPE IDENTIFIER discriminant_part IS PRIVATE ';'
        { private_type_declaration.generic_type_declarations =
          [IDENTIFIER.%text," : GENERIC_TYPE\n"];
         private_type_declaration.type_declarations =
          [" ",IDENTIFIER.%text," : ADT\n"]; }
    | TYPE IDENTIFIER discriminant_part IS LIMITED PRIVATE ';'
        {}
    ;

incomplete_type_declaration
    : TYPE IDENTIFIER discriminant_part ';'
        {}
```

```
    ;

deferred_constant_declaration
    : identifier_list ':' CONSTANT name ';'
        {}
    ;

attribute
    : prefix TIC attribute_designator
        {}
    ;

attribute_designator
    : IDENTIFIER
        {}
    | IDENTIFIER '(' expression ')'
        {}
    ;

expression
    : relation
        {}
    | relation and_relations
        {}
    | relation or_relations
        {}
    | relation xor_relations
        {}
    | relation and_then_relations
        {}
    | relation or_else_relations
        {}
    |
        {}
    ;

and_relations
    : and_relations AND relation
        {}
    |
        {}
    ;

or_relations
    : or_relations OR relation
        {}
    |
        {}
    ;

xor_relations
    : xor_relations XOR relation
        {}
    |
        {}
    ;

and_then_relations
    : and_then_relations OR relation
        {}
    |
```

```
        {}
    ;

or_else_relations
    : or_else_relations OR relation
        {}
    |
        {}
    ;

relation
    : simple_expression
        {}
    | simple_expression relational_operator simple_expression
        {}
    | simple_expression IN range
        {}
    | simple_expression NOT IN range
        {}
    | simple_expression IN name
        {}
    | simple_expression NOT IN name
        {}
    ;

simple_expression
    : term  !Not really required because 3rd choice can break down to term
        {}
    | unary_adding_operator term
        {}
    | term binary_adding_operator binary_terms
        {}
    | unary_adding_operator binary_terms
        {}
    ;

binary_terms
    : binary_terms binary_term
        {}
    | binary_term
        {}
    ;

binary_term
    : term
        {}
    | term binary_adding_operator
        {}
    ;

term
    : factor multiplying_operator_factor
        {}
    ;

multiplying_operator_factor
    : multiplying_operator_factor multiplying_operator factor
        {}
    |
        {}
    ;
```

```
multiplying_operator
    : TIMES
        {}
    | DIVIDE
        {}
    | MOD
        {}
    | REM
        {}
    ;

factor
    : primary
        {}
    | highest_precedence_operator primary
        {}
    | primary highest_precedence_operator primary
        {}
    ;

highest_precedence_operator
    : EXPONENT
        {}
    | ABS
        {}
    | NOT
        {}
    ;

primary
    : numeric_literal
        {}
    | NULL
        {}
    | aggregate
        {}
    | STRING_LITERAL
        {}
    | name
        {}
    | allocator
        {}
    | function_call
        {}
    | type_conversion
        {}
    | qualified_expression
        {}
    | '(' expression ')'
        {}
    ;

numeric_literal
    : INTEGER_LITERAL
        {}
    | REAL_LITERAL
        {}
    ;

aggregate
```

```
    : '(' component_associations ')'
        {}
    ;

component_associations
    : component_associations ',' component_association
        {}
    | component_association
        {}
    ;

component_association
    : expression
        {}
    | choices ARROW expression
        {}
    ;

choices
    : choices '|' choice
        {}
    | choice
        {}
    ;

allocator
    : NEW qualified_expression
        {}
    | NEW subtype_indication
        {}
    ;

function_call
    : name
        {}
    | name actual_parameter_part
        {}
    ;

actual_parameter_part
    : '(' parameter_associations ')'
        {}
    ;

parameter_associations
    : parameter_associations ',' parameter_association
        {}
    | parameter_association
        {}
    ;

parameter_association
    : IDENTIFIER ARROW actual_parameter
        {}
    | actual_parameter
        {}
    ;

actual_parameter
    : expression
        {}
```

```
    | name '(' name ')'
        {}
    | name
        {}
    ;

type_conversion
    : name '(' expression ')'
        {}
    ;

qualified_expression
    : name TIC '(' expression ')'
        {}
    | name TIC aggregate
        {}
    ;

relational_operator
    : EQUAL
        {}
    | NEQ
        {}
    | LT
        {}
    | LTE
        {}
    | GT
        {}
    | GTE
        {}
    ;

binary_adding_operator
    : PLUS
        {}
    | MINUS
        {}
    | AND  !&, string concatenation
        {}
    ;

unary_adding_operator
    : PLUS
        {}
    | MINUS
        {}
    ;

multiplying_operator
    : TIMES
        {}
    | DIVIDE
        {}
    | MOD
        {}
    | REM
        {}
    ;

enumeration_type_definition
```

```
    : '(' enumeration_literal_specifications ')'
        {}
    ;

enumeration_literal_specifications
    : enumeration_literal_specifications ',' enumeration_literal
        {}
    | enumeration_literal
        {}
    ;

enumeration_literal
    : IDENTIFIER
        {}
    | CHARACTER_LITERAL
        {}
    ;

integer_type_definition
    : range_constraint
        {}
    ;

real_type_definition
    : floating_point_constraint
        {}
    | fixed_point_constraint
        {}
    ;

floating_point_constraint
    : floating_accuracy_definition
        {}
    | floating_accuracy_definition range_constraint
        {}
    ;

floating_accuracy_definition
    : DIGITS simple_expression
        {}
    ;

fixed_point_constraint
    : fixed_accuracy_definition
        {}
    | fixed_accuracy_definition range_constraint
        {}
    ;

fixed_accuracy_definition
    : DELTA simple_expression
        {}
    ;

array_type_definition
    : unconstrained_array_definition
        { array_type_definition.variable_type =
          unconstrained_array_definition.variable_type;
         unconstrained_array_definition.generic_types =
          array_type_definition.generic_types;
         array_type_definition.new_composite_types =
```

```
            unconstrained_array_definition.new_composite_types; }
      | constrained_array_definition
          { array_type_definition.variable_type =
            constrained_array_definition.variable_type;
           constrained_array_definition.generic_types =
            array_type_definition.generic_types;
           array_type_definition.new_composite_types =
            constrained_array_definition.new_composite_types; }
      ;

unconstrained_array_definition
      : ARRAY '(' subtype_definitions ')' OF subtype_indication
          { unconstrained_array_definition.variable_type =
            ["GENERIC_TYPE[BASE_TYPE: ARRAY[ARRAY_ELEMENT:",
             subtype_indication.element_type,",\n\t\t\tARRAY_INDEX:",
             subtype_definitions.index_type,"]]"];
           subtype_definitions.generic_types =
            unconstrained_array_definition.generic_types;
           subtype_indication.generic_types =
            unconstrained_array_definition.generic_types;
           unconstrained_array_definition.new_composite_types =
            subtype_definitions.new_composite_types +|
            subtype_indication.new_composite_types; }
      ;

subtype_definitions
      : subtype_definitions ',' index_subtype_definition
          { index_subtype_definition.generic_types =
            subtype_definitions.generic_types;
           subtype_definitions.index_type =
            index_subtype_definition.index_type;
           subtype_definitions[1].new_composite_types =
            subtype_definitions[2].new_composite_types +|
            index_subtype_definition.new_composite_types; }
      | index_subtype_definition
          { index_subtype_definition.generic_types =
            subtype_definitions.generic_types;
           subtype_definitions.index_type =
            index_subtype_definition.index_type;
           subtype_definitions.new_composite_types =
            index_subtype_definition.new_composite_types; }
      ;

constrained_array_definition
      : ARRAY index_constraint OF subtype_indication
          { constrained_array_definition.variable_type =
            ["GENERIC_TYPE[BASE_TYPE: ARRAY[ARRAY_ELEMENT:",
             subtype_indication.element_type,
             "\n\t\t\tARRAY_INDEX:DISCRETE]]"];
           subtype_indication.generic_types =
            constrained_array_definition.generic_types;
           constrained_array_definition.new_composite_types =
            subtype_indication.new_composite_types; }
      ;

index_subtype_definition
      : name RANGE DISCRETE
          { name.generic_types = index_subtype_definition.generic_types;
           index_subtype_definition.index_type = "DISCRETE";
           index_subtype_definition.new_composite_types =
            name.new_composite_types; }
```

```
    ;

index_constraint
    : '(' discrete_ranges ')'
        {}
    ;

discrete_ranges
    : discrete_ranges ','discrete_range
        {}
    | discrete_range
        {}
    ;

discrete_range
    : subtype_indication
        {}
    | range
        {}
    ;

record_type_definition
    : RECORD component_list END RECORD
        {}
    ;

component_list
    : component_declarations
        {}
    | component_declarations variant_part
        {}
    | variant_part
        {}
    | NULL ';'
        {}
    ;

component_declarations
    : component_declarations component_declaration
        {}
    | component_declaration
        {}
    ;

component_declaration
    : identifier_list ':' subtype_indication ';'
        {}
    | identifier_list ':' subtype_indication EQUAL expression ';'
        {}
    ;

discriminant_part
    : '(' discriminant_specifications ')'
        {}
    ;

discriminant_specifications
    : discriminant_specifications ';' discriminant_specification
        {}
    | discriminant_specification
        {}
```

```
    ;

discriminant_specification
    : identifier_list ':' name
      {}
    | identifier_list ':' name EQUAL expression
      {}
    ;

variant_part
    : CASE IDENTIFIER IS variants END CASE ';'
      {}
    ;

variants
    : variants variant
      {}
    | variant
      {}
    ;

variant
    : WHEN choices ARROW component_list
      {}
    ;

choice
    : simple_expression
      {}
    | discrete_range
      {}
    | OTHERS
      {}
    | IDENTIFIER
      {}
    ;

access_type_definition
    : ACCESS subtype_indication
      {}
    ;

name
    : IDENTIFIER
      { name.variable_type = IDENTIFIER.%text;
        name.variable_name = IDENTIFIER.%text;
        name.element_type = name.generic_types(IDENTIFIER.%text);
        name.index_type = name.generic_types(IDENTIFIER.%text);
        name.new_composite_types = {(IDENTIFIER.%text: "yes")}; }
    | CHARACTER_LITERAL
      {}
    | STRING_LITERAL      !operator_symbol
      {}
    | indexed_component
      {}
    | slice
      {}
    | selected_component
      {}
    | attribute
      {}
```

```
    ;

prefix
    : name
      {}
    | function_call
      {}
    ;

indexed_component
    : prefix '(' expressions ')'
      {}
    ;

expressions
    : expressions ',' expression
      {}
    | expression
      {}
    ;

slice
    : prefix '(' discrete_range ')'
      {}
    ;

selected_component
    : prefix '.' selector
      {}
    ;

selector
    : ALL
      {}
    | CHARACTER_LITERAL
      {}
    | STRING_LITERAL      !operator_symbol
      {}
    | IDENTIFIER          !simple_name
      {}
    ;

identifier_list
    : identifier_list ',' IDENTIFIER
      { identifier_list[1].variable_names =
        [identifier_list[2].variable_names,", ",IDENTIFIER.%text]; }
    | IDENTIFIER
      { identifier_list.variable_names = IDENTIFIER.%text; }
    ;
```

LIST OF REFERENCES

[1] Booch, G., "*Software Components with Ada, Structures, Tools, and Subsystems*", The Benjamin/Cummings Publishing Company, 1987.

[2] Booch, G., "*Software Engineering with Ada*", 2nd ed., The Benjamin/Cummings Publishing Company, 1987.

[3] DePasquale, G., "*Design and Implementation of Module Driver and Output Analyzer Generator*", Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1990.

[4] Herndon, R., "*The Incomplete AG User's Guide and Reference Manual*", University of Minnesota Computer Science Technical Report 85-37, October 1985.

[5] Jones, C., "Reusability in Programming: A Survey of the State of the Art", *IEEE Transactions on Software Engineering*, September 1984, Vol. SE-10 (5).

[6] Li, H., van Katwijk, J., "Issues Concerning Software Reuse-in-the-Large", in *Proceedings of the Second International Conference on Systems Integration*, Morristown, NJ, June 1992, pp. 66-75.

[7] Luqi, "Computer-Aided Prototyping for a Command-And-Control System Using CAPS", *IEEE Software*, January 1992, pp. 56-67.

[8] Luqi, "Real-Time Constraints in a Rapid Prototyping Language", *Journal of Computer Languages*, Spring 1991, Vol. 18, No. 2, pp. 77-103.

[9] Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer*, May 1989, pp. 13-25.

[10] Luqi, Berzins, V., Yeh, R., "A Prototyping Language for Real-Time-Software", *IEEE Transactions on Software Engineering*, October 1988, Vol. 14, No. 10, pp. 1409-1423.

[11] Luqi, Lee, Y., "Towards Automated Retrieval of Reusable Software Components", in Workshop Notes of *the AAAI Workshop on Artificial Intelligence and Automated Program Understanding*, San Jose, CA, July 13, 1992, pp. 85-88.

[12] Luqi, McDowell, J., "Software Reuse in Specification-Based Prototyping", in *Proceedings of the 14th Annual Software Reuse Workshop*, Herndon, VA, November 18-22, 1991, pp. 1-7.

[13] Luqi, Steigerwald, R., Hughes, G., Naveda, F., Berzins, V., "CAPS as a Requirements Engineering Tool", in *Proceedings of Requirements Engineering and Analysis Workshop*, Software Engineering Institute, Carnegie Mellon University, March 12-14, 1991, Pittsburgh, PA, pp. 1-8.

[14] McDowell, J., "*A Reusable Component Retrieval System for Prototyping*", Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

[15] Rachal, Randy J., "*Design and Implementation of a Concrete Interface Generation System*", Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

[16] Steigerwald, R., "*Reusable Software Component Retrieval via Normalized Algebraic Specifications*", Ph.D. Dissertation, Naval Posgraduate School, Monterey, CA, December 1991.

[17] Steigerwald, R., Luqi, Berzins, V., "A Tool for Reusable Software Component Retrieval via Normalized Specifications", in *Proceedings of the Hawaii Conference on System Sciences*, Koloa, Hawaii, January 7-10, 1992, pp. 18-26.

[18] Steigerwald, R., Luqi, McDowell, J., "CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping", *Information and Software Technology*, England, November 1991, Vol. 38, No. 11.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman Code CS, Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5100	1
4.	Dr. Yuh-jeng Lee Code CS/LE, Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5100	8
5.	Dr. Luqi Code CS/LQ, Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5100	1
6.	Lieutenant Jennie M. Sealander COMFAIRMED (N8) PSC 810, Box 2 FPO AE 09619-2000	1

Thesis
S40523 Sealander
c.1 Building reusable
software components for
automated retrieval.